Programming Concept & Paradigms

Zusammenfassung Modul PCP HSLU.I FS20

Pascal Kiser pascal.kiser@stud.hslu.ch

29.02.2020

Inhaltsverzeichnis

1 Einführung Programmierparadigmen

Wikipedia listet etwa 700 “namhafte” Programmiersprachen auf.

Zusätzlich existieren tausende weitere, spezialisierte Sprachen.

1.1 Wichtige Begriffe

1.2 Programmiersprache

Eine *Programmiersprache* ist eine formal konstruierte Sprache, entworfen um Befehle an Maschinen, speziell Computer, zu übermitteln. Programmiersprachen werden eingesetzt, um Programme zu schreiben, welche das Verhalten von Maschinen kontrollieren oder welche Algorithmen beschreiben.

Ein *Algorithmus* ist ein schrittweises Verfahren zur Lösung von einem Problem oder einer Klasse von Problemen.

Syntax gibt die formale Struktur vor, nach dem Programme einer Sprache aufgebaut sind.

Semantik definiert die Bedeutung von Programmen (Anweisungen, Operatoren,...), entspricht also den “Interpretationsregeln”.

1.2.1 Mächtigkeit

Alle Programmiersprachen sind im Prinzip gleich mächtig, d.h. in diesen können dieselben Algorithmen festgehalten werden.

In der Berechenbarkeitstheorie spricht man von *Turing-Vollständigkeit* (= universelle Programmierbarkeit), wenn eine Programmiersprache grundsätzlich alle Funktionen berechnen kann, die mit einer Turing-Maschine berechnet werden können. Dies ist eine notwendige Bedingung für eine Programmiersprachen.

Gemäss dieser Definition ist z.B. HTML keine Programmiersprache, SQL seit der Einführung von rekursiven Common-Table-Expression mit SQL 1999 aber schon.

1.2.2 Programmierparadigma

Ein *Paradigma* im Allgemeinen beschreibt ein (Denk-)Muster, eine grundlegende Denkweise oder Weltanschauung.

Ein *Programmierparadigma* ist ein fundamentaler Programmierstil, eine bestimmt Art die Struktur und Elemente von Programmen aufzubauen.

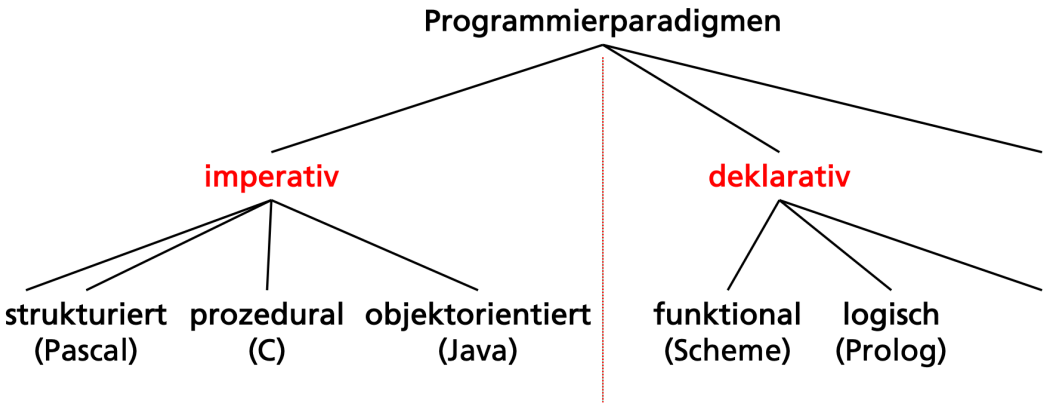

Abbildung 1: Paradigmenbaum

Fundamental ist vor allem die Unterscheidung in *imperative* und *deklarative* Programmiersprachen.

Weitere Paradigmen sind beispielsweise:

- Aspekt-orientiert
- Automaten-basiert
- Constraint-orientiert

- Daten-getrieben
- Datenfluss-orientiert
- Ereignis-orientiert,
- generisch
- generativ
- Komponenten-orientiert
- nebenläufig
- modular
- Protokoll-orientiert
- reaktiv
- reflektiv
- ...

Siehe auch: Wikipedia

Viele dieser Paradigmen lassen sich in verschiedenen Sprachen umsetzten bzw. integrieren. Diese Paradigmen sind daher typischerweise nicht prägend für eine bestimmte Programmiersprache.

Wir konzentrieren uns deshalb auf die Programmierparadigmen *imperativ* (strukturiert, prozedural, objektorientiert) und *deklarativ* (logisch, funktional).

1.3 Sprachen und Paradigmen

Die meisten Programmiersprachen haben ein *Hauptparadigma*, das sie befolgen, z.B.:

- Java: objektorientiert
- Prolog: logisch
- Scheme: funktional

Diese Unterscheidung ist nicht messerscharf: objektorientierte Sprachen sind beispielsweise meist auch strukturiert und prozedural und die meisten deklarativen Sprachen haben auch imperative Eigenschaften.

Sprachen müssen nicht exklusiv einem Programmierparadigma angehören. Programmierparadigmen sind eher komplementär als sich gegenseitig ausschliessend. Viele moderne Sprachen sind *Multiparadigmen-Programmiersprachen*, d.h. sie unterstützen mehrere Paradigmen.

So ist beispielsweise Java zwar primär objektorientiert mit prozeduralen, modularen, ereignisorientierten und (seit Java 8) auch funktionalen Ansätzen.

1.4 Imperativ vs Deklarativ

Imperative Programmierung beschreibt primär das *Wie*:

Die Problemlösung (Berechnungslogik) wird schrittweise in Befehlen beschrieben: “Wie ist das Problem zu lösen?”

Bespiel Kreis (imperativ):

Resultat einer 360 Grad Rotation um einen festen Mittelpunkt mit dem Zirkel

Deklarative Programmierung beschreibt primär das *Was*:

Berechnungslogik wird beschrieben ohne den Kontrollfluss zu definieren: “Was sind die Fakten zum Problem?”

1.4.1 Beispiel Kreis

Aufgabe: Zeichne einen Kreis.

- **Imperativ**: Resultat einer 360 Grad Rotation um einen festen Mittelpunkt mit dem Zirkel
- **Deklarativ**: Menge aller Punkte, die von einem vorgegebenen Punkt denselben Abstand hat.

1.4.2 Beispiel GCD

Aufgabe: Berechnung des grössten gemeinsamen Teilers (*greatest common divisor: GCD*) von zwei ganzen Zahlen gemäss dem euklidischen Algorithmus.

Hinweis: Wir gehen davon aus, dass die beiden Zahlen positive Ganzzahlen sind (ohne dies zu testen).

Imperativ / prozedural: (C):

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b) a = a -b;
        else b = b - a;
    }
    return a;
}
```

Der Fokus ist auf den Befehlen, der Kontrollfluss ist vorgegeben (sequentiell).

Deklarativ-logisch (Prolog):

```
gcd(A, A, A) .
gcd(A, B, G) :- A > B, C is A-B, gcd(C, B, G).
gcd(A, B, G) :- B > A, C is B-A, gcd(C, A, G).
```

Einzelne Fakten und Regeln sind gegeben, aber der Kontrollfluss ist nicht (explizit) vorgegeben.

Aufruf in der Prolog-Konsole:

```
?- gcd(42, 35, X) .
X = 7
```

Deklarativ-funktional (Scheme):

```
(define gcd
    (lambda (a b)
        (cond   ((= a b) a)
                ((> a b) (gcd( (- a b) b) ) )
                (else (gcd (- b a) a)))))
```

Fokus auf Definition von Ein- und Ausgaben von Funktionen. Der Kontrollfluss ist nicht (explizit) vorgegeben.

Aufruf in der Scheme-Konsole:

```
> (gcd 42 35)
7
```

1.5 Imperative Programmierung

> imperare (lat) = befehlen, anordnen

Der Problemlösungs-Algorithmus wird schrittweise durch Befehle angegeben. Der Quellcode gibt an, was der Computer in welcher Reihenfolge tun soll.

Zur Steuerung der Befehlsausführung stehen Kontrollstrukturen (Sequenz, Iteration & Selektion) zur Verfügung. Die meisten populären Programmiersprachen sind imperativ (Java, C#, PHP). Insbesondere funktionieren Hardware-nahe Sprachen wie Assembler oder C praktisch immer so.

Das Imperative Programmierparadigma ist eng angelehnt an die Ausführung von Maschinen-Code auf Computern, die nach der Von-Neumann-Architektur implementiert sind:

- Es gibt bedingte und unbedingte Sprunganweisungen

- Zustand vom Programm ergibt sich aus Inhalt von Datenfeldern im Arbeitsspeicher und Systemvariablen (Register, Befehlszähler)

Die Von-Neumann-Architektur realisiert alle Komponenten einer Turing-Maschine und ist das wichtigste Referenzmodell für Computer.

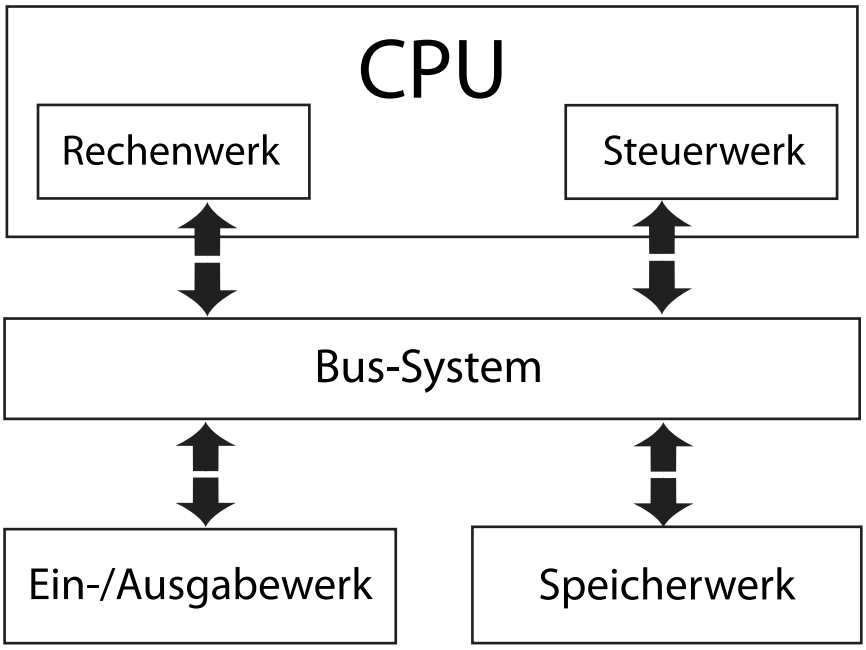

Abbildung 2: Von-Neumann-Architektur

https://upload.wikimedia.org/wikipedia/commons/d/db/Von-Neumann_Architektur.svg

1.5.1 Strukturierte Programmierung

Die *strukturierte Programmierung* ist eine Spezialisierung des imperativen Paradigmas, und verlangt insbesondere die Beschränkung auf drei Kontrollstrukturen:

- Sequenzen
- Auswahl
- Wiederholung

Die Konsequenz daraus ist, dass `GOTO` nicht verwendet werden darf. Typische strukturierte Sprachen sind *C*, *Fortan* & *Pascal*.

Lesenswerter Artikel:

„Go To Statement Considered Harmful“ von Edsger Dijkstra (Com. of the ACM, Vol. 11, No. 3, March 1968):

- http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html

1.5.2 Prozedurale Programmierung

Die *prozedurale Programmierung* ist eine Spezialisierung des imperativen Paradigmas, welches die Unterteilung von Programmen in Teilprogramme (Prozedur, Routine, Funktion, ...) bietet.

Eigenschaften von Prozeduren:

- können Argumente entgegen nehmen und Ergebnisse zurückgeben
- Vermeidung von Code-Duplikation
- Unterscheidung von Programm-globalen und Prozedur-lokalen Variablen

Typische strukturierte Sprachen sind *C*, *Fortan* & *Pascal*.

1.5.3 Objektorientierte Programmierung

Die *objektorientierte Programmierung* ist eine Spezialisierung des imperativen Paradigmas. Laufende Programme bestehen aus einzelnen Objekten, welche miteinander interagieren (Nachrichten austauschen, Methoden aufrufen) können.

Objekte sind typischerweise Instanzen von Klassen, aber es gibt auch andere Ansätze (z.B. Prototypen bei JavaScript).

Typische Eigenschaften von Objektorientierung mit Klassen:

- Klasse definiert Zustand (Variablen) & Verhalten (Methoden)
- Vererbung
- Polymorphismus

Die objektorientierte Programmierung als Weiterentwicklung von prozeduraler Programmierung und abstrakten Datentypen. Typische Sprachen sind:

- *Smalltalk*
- *Objective C*
- *C++*
- *Java*
- *C#*

1.6 Deklarative Programmierung

declaration (lat) = Erklärung, Kundmachung, Offenbarung

Das zu lösende Problem wird beschrieben, der Lösungsweg wird dann automatisch ermittelt: Das Programm enthält genügend Information, sodass das Problem gelöst werden kann. Der Kontrollfluss von einem Programm ist nicht (explizit durch den Programmierer) geregelt.

Bekannteste Unter-Programmierparadigmen sind logische Programmierung (z.B. Prolog) und funktionale Programmierung (z.B. Scheme). Weitere deklarative Programmierparadigmen sind z.B.: Constraint Programming, Abfragesprachen (z.B. SQL), oder Datenflusssprachen (z.B. Simulink).

1.6.1 Logische Programmierung

Die *logische Programmierung* ist eine Spezialisierung des deklarativen Paradigmas. Das Programm besteht aus FAkten und Regeln, aus welchen auf Anfrage automatisch versucht wird, eine Lösungsaussage herzuleiten. Der Lösungsweg wird nicht angegeben.

Logische Programmierung basiert auf mathematischer Logik (*Horn-Klauseln Resolution*). Bekannteste Sprachen sind *Prolog* und *Datalog*.

1.6.2 Funktionale Programmierung

Die *funktionale Programmierung* ist eine Spezialisierung des deklarativen Paradigmas. Programme bestehen ausschliesslich aus Definitionen von Funktionen mit Parametern und Rückgabewerten.

Eigenschaften:

- Rückgabewert hängt ausschliesslich von den Parametern ab (*referenzielle Transparenz*)
- Es gibt keinen veränderbaren Zustand und keine veränderbaren Daten

Die logische Programmierung basiert auf dem formalen System des *Lambda-Kalkül*. Bekannte Sprachen sind *Clojure*, *Erlang*, *Haskell*, *Lisp*, *ML* & *Scheme*. # Imperative Sprachen am Beispiel C & Java

Die Programmiersprachen C und Java sind beide *imperativ*, *strukturiert* und *prozedural*. Java ist zusätzlich *objektorientiert*.

Java basiert auf C und die Syntax und Semantik sind weitgehend identisch:

- Gemeinsame Schlüsselwörter (`char`, `double`, `enum`, `if`, ...)
- Gleiche Klammerung von Blöcken (`{` und `}`)
- Gleiche Syntax und Semantik für die zentralen Kontrollstrukturen (if, if-else, for- & while-Schleifen)
- Gleiche elementare Datentypen (int, char, float, ...)

1.7 Datentypen in C

C kennt elementare Datentypen wie int, char oder double und die Datenstruktur Array. Zusammengesetzte Datentypen können mit Hilfe von Strukturen (`struct`) und Typendefinitionen (`typedef`) erzeugt werden.

Beispiel:

```
typedef struct {
    int x;
    int y;
    char color;
} point_type;

point_type pt;
pt.x = 100;
pt.y = 200;
```

1.8 Abstrakte Datentypen

Abstrakte Datentypen (ADT) sind mathematische Modelle für Datenstrukturen und darauf definierte Operationen (mit vorgegebenem Effekt). Die Semantik ist vorgegeben, die Implementierung nicht.

Typische Beispiele für ADTs:

Name	Operationen
Stack	init, push, pop, top
Queue	init, add, remove
Matrix	addition, multiplication, inverse...

1.9 Von ADT zu OOP

ADT bieten Datenstrukturen und darauf klar definierte Operationen (resp. Funktionen). Die Implementierung der Datenstruktur ist grundsätzlich getrennt und

unabhängig von der Implementierung der dazugehörigen Funktionen. Eine der Hauptmotivationen für die Objektorientierung war das zusammenbringen von Daten und Operationen.

Klassen haben definierte Zustände (Daten, Instanzvariablen) und Verhalten (Methoden). Die Funktionen "hängen" direkt and den Daten. In diesem Sinne kann OO als eine Weiterentwicklung von ADTs verstanden werden.

1.10 Packages und Sichtbarkeit

Mit der Definition von Klassen und Packages (hierarchischen Gruppen von Klassen) wurden auch Fragen der Sichtbarkeit relevant(er): Von wo aus kann auf was (Methode / Variable) zugegriffen werden?

Java verwendet dafür eine vierstufige Sichtbarkeit, deklariert durch die vier Zugriffsmodifizierer:

- `private`
- `default`
- `protected`
- `public`

Java erweitert C im wesentlichen um folgende Konzepte:

- Klassen und Instanzen
- Interfaces
- Packages
- Sichtbarkeit
- Exception Handling

1.10.1 ADT Beispiel: Stack

Ein Stack ist eine Sammlung von Elementen mit bestimmter Struktur. Neue Elemente könnten hinzugefügt werden (Operation `push`), das zuletzt hinzugefügte Element kann abgefragt werden (`top`) und mit `pop` vom Stack entfernt werden.

LIFO-Prinzip: Last-in-first-out

1.10.2 Implementierung in C

Einfache Implementation eines Stacks in C. Typendefinitionen:

```
#define STACK_SIZE 3
#define STACK_EMPTY_INDEX -1
#define STACK_DUMMY_ELEMENT -1337 // arbitrary number

typedef int element;
typedef struct {
    element stackArray[STACK_SIZE];
    int index;
} stack;
```

Signaturen der Stack-Operationen:

```
stack init(); // returns new empty stack
stack push(element e, stack s); // adds element to stack
element top(stack s); // returns top element
stack pop(stack s); // removes top element
void print(stack s); // prints all elements
```

Implementation von `init()`:

```
// returns new empty stack
stack init() {
    stack s;
    s.index = STACK_EMPTY_INDEX;
    return s;
}
```

Implementation von `push(...)`:

```
// adds element to stack
stack push(element e, stack s) {
  if (s.index + 1 < STACK_SIZE) {
    // there is space for one more element
    s.index++;
    s.stackArray[s.index] = e;
  } else {
    printf("ERROR - push: stack full! Cannot add %i\n", e);
  }
  return s;
}
```

Implementation von `top(...)`:

```
// returns top element
element top(stack s) {
  if (s.index > STACK_EMPTY_INDEX) {
    return s.stackArray[s.index];
  } else {
    printf("ERROR - top: stack empty!\n");
    return STACK_DUMMY_ELEMENT;
  }
}
```

Implementation von `pop(...)`:

```
// removes top element from stack
stack pop(stack s) {
  if (s.index > STACK_EMPTY_INDEX) {
    s.index--;
  }
  return s;
}
```

Bemerkungen:

Strukturen werden in C als Werte (*call by value*) übergeben. Wenn eine Referenz übergeben werden soll, muss das mit Pointern gemacht werden. Auf die Verwendung von Pointern und verketteten Listen wurde in diesem Beispiel der Einfachheit halber verzichtet. Aus dem gleichen Grund wurde int als Datentyp für die Elemente gewählt. Grundsätzlich hätte hier jeder beliebige Typ gewählt werden können.

1.10.3 Implementierung in Java

In Java können die Daten und die Operationen des ADT Stack in einer Klasse zusammengefasst werden. Stack-Operationen können mit Hilfe entsprechender Instanz-Methoden direkt auf Stack-Instanzen aufgerufen werden:

```
Stack myStack = new Stack();
myStack.top();
myStack.push(new Element(42));
```

Statt wie vorher in C:

```
stack myStack = init();
top(myStack);
myStack = push(42, myStack);
```

2 Prolog

Wir verwenden `swi-prolog`: http://www.swi-prolog.org

Prolog (*PROgrammation en LOGique*) entstand in den frühen 1970er Jahren und basiert auf der Prädikatenlogik erster Stufe. Die Notation von Regeln entspricht Horn-Klauseln.

2.1 Big Picture: Funktionsweise von Prolog

Wissensdatenbank bestehend aus Fakten und Regel. Diese kann abgefragt werden durch Queries.

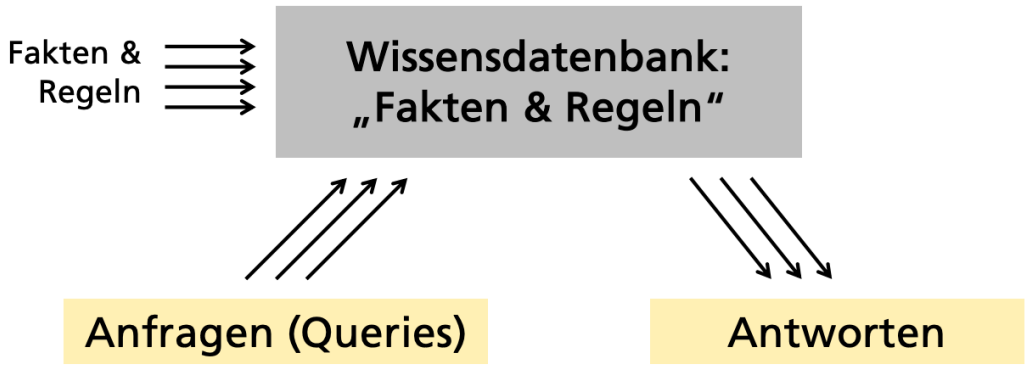

Abbildung 3: Funktionsweise von Prolog

2.2 Einstiegsbeispiel

Ein erstes Prolog-Programm, bestehend aus den folgenden drei Beispiel-Fakten in der Wissensdatenbank:

```
bigger(elephant, horse).
bigger(horse, dog).
bigger(horse, sheep).
```

Bemerkung: `bigger` ist nicht vordefiniert vom System. Die Definitionen haben für Prolog keine Bedeutung, und weiss beispielsweise nicht, dass die Eigenschaft `bigger` transitiv sein sollte. Deshalb beantwortet Prolog folgende Abfrage mit `false`:

```
?- bigger(elephant, dog).
false.
```

Deshalb definieren wir eine *transitive Hülle*, d.h. ein Prädikat, mit welchem ein Tier A grösser ist als ein Tier B, wenn wir dazwischen mit den früher definierten Fakten über andere Tiere iterieren können.

```
is_bigger(X, Y) :- bigger(X, Y).
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).
```

- `:-` = `if`
- `,` = `and`

Mit dieser neuen Abfrage bekommen wir dieses Resultat:

```
?- is_bigger(elephant, dog).
true.
```

Die selbe Abfrage kann mit einer Variable verwendet werden. Die Ausgabe enthält dann alle gültigen Werte für diesen Ausdruck:

```
?- is_bigger(X, dog).
X = horse ;
X = elephant ;
false.
```

Eine weitere Beispiel-Abfrage:

Gibt es ein Tier, das kleiner als ein Elefant und grösser als ein Hund ist?

```
?- is_bigger(elephant, X), is_bigger(X, dog).
X = horse ;
false.
```

2.3 Prolog Syntax

2.3.1 Terme

Zahlen (`numbers`):

```
123, 33.3, -4
```

Atomare Terme (`atoms`): beginnen mit Kleinbuchstaben oder sind eingeschlossen in einfache Anführungszeichen:

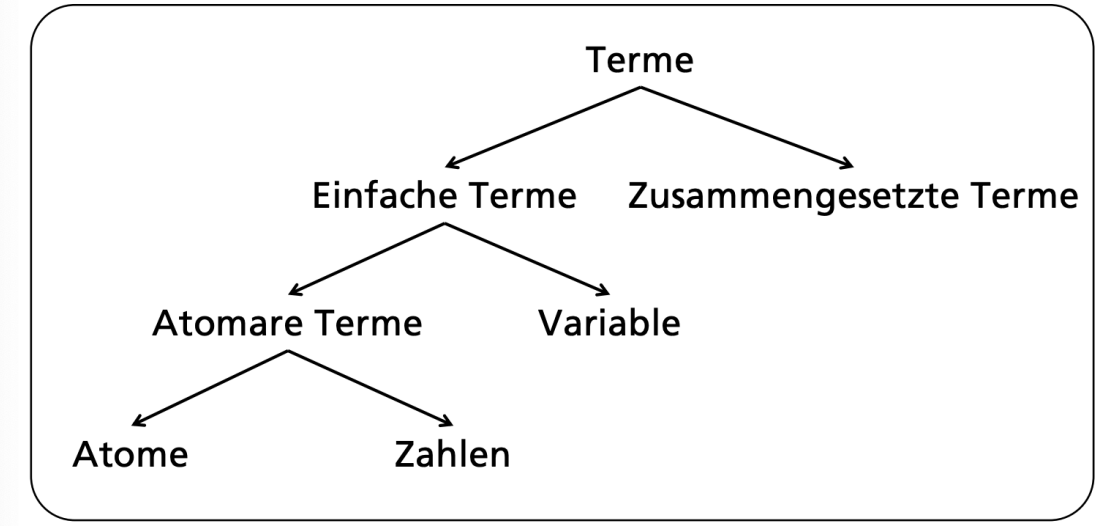

Abbildung 4: Prolog basic syntax

```
elephant, a_bc, ‘Hallo mein Text‘, is_bigger
```

Variablen (`variables`): beginnen mit Grossbuchstaben oder einem Unterstrich (underscore)

```
X, Elephant, _, _elephant
```

Anonyme Variablen (_): Platzhalter für einen Wert, der nicht ausgegeben werden soll:

```
?- is_bigger(_, horse).
true .
```

Zusammengesetzte Terme: (`compound terms`): besteht aus einem Funktor und Argumenten:

```
functor(argument1, argument2)
```

Aus dem Beispiel oben:

```
is_bigger(horse, X)
```

2.3.2 Grundterme

Terme ohne Variablen, 'Fakten' in der Wissensdatenbank:

```
bigger(me, you)
write('bonjour monde')
```

2.4 Prädikate

Die *Stelligkeit* eines Prädikats beschreibt die Anzahl Argumente. Prolog behandelt zwei Prädikate mit gleichem Funktor aber mit unterschiedlicher Stelligkeit als zwei verschiedene Prädikate.

n der Prolog-Doku wird die Stelligkeit meist mit Suffix „/" und danach der entsprechenden Zahl angegeben:

- `consult/1`
- `working_directory/2`
- `is_bigger/2`

2.4.1 Eingebaute Prädikate

SWI-Prolog enthält zahlreiche eingebaute Prädikate. Diese sind in derOnline Doku beschrieben: http://www.swi-prolog.org/pldoc/man?section=builtin

Beispiele:

1. Eine Programm-Datei kompilieren: `consult/1`

```
?- consult(bigger).
% bigger.pl compiled 0.00 sec, 12 clauses true.
?-
```

Liest die Datei `bigger` oder `bigger.pl` ein. Gegebenenfalls muss davor mit `working_directory/2` das aktuelle Verzeichnis gesetzt werden.

2. Argumente in die Konsole schreiben: `write/1`

```
?- write('hello world').
hello world
true.
```

3. Standard Input lesen: `read/1`

```
?- read(X), write(X).
|: 'hello world'.
hello world
X = 'hello world'.
```

2.5 Prolog Queries

Anfragen (Queries) sind Prädikate (oder Sequenzen von Prädikaten) gefolgt von einem Punkt. Diese werden beim Prolog-Prompt (`?-`) in der Konsole eingegeben und veranlassen das System zu antworten:

```
?- is_bigger(elephant, dog). % query
true .
```

2.6 Wichtige Begriffe

- *Klauseln* (clauses) = Fakten und Regeln (sind zusammengesetzte Terme)
- *Prozeduren* (procedure) = Alle Klauseln zum gleichen Prädikat (d.h. alle Relationen mit gleichem Name [d.h gleichem Funktor] & gleicher Stelligkeit)
- Prolog-Programm = eine Liste von Klauseln

Prolog-Programme bestehen aus Fakten und Regeln, die in der aktuellen Wissensdatenbank abgelegt sind.

- *Fakten* (facts) = Prädikate gefolgt von einem Punkt. Fakten sind typischerweise Grundterme (Prädikate ohne Variablen)
- *Regeln* (rules) bestehen aus einem Kopf (head) und einem Hauptteil (body), durch `:-` getrennt. Der Kopf einer Regel ist wahr,falls alle Ziele (Prädikate) im Hauptteil als wahr bewiesen werden können, z.B.:

```
grandfather(X, Y) :-          % head
    father(X, Z),              % body, goal 1
    parent(Z, Y).              % body, goal 2
```

Ziele im Hauptteil werden durch `,` (Komma) abgetrennt und ganz am Schluss durch `.` (Punkt) abgeschlossen.

2.7 Matching

Zwei atomare Terme *matchen* genau dann wenn sie die gleiche Zahl oder das gleiche Atom sind. Falls einer der Terme eine Variable ist, dann matchen die beiden Terme und die Variable wird mit dem Wert des zweiten Terms instanziiert.

Zwei zusammengesetzte Terme matchen genau dann, wenn:

– gleicher Funktor, gleiche Stelligkeit – alle korrespondierenden Argumente matchen

2.7.1 Beispiel

Zwei Regeln in der Wissensdatenbank:

```
vertical(line(point(X,_), point(X,_))).
horizontal(line(point(_,Y), point(_,Y))).
```

Beispielabfragen:

```
?- vertical(line(point(1,1), point(1,5))).
true.
?- horizontal(line(point(1,2),point(3,X))).
X = 2.
?- horizontal(line(point(1,2),P)).
P = point(_G1155, 2). % _G1155 means "any number"
```

2.8 Beweissuche und Suchbäume

Bei einer Anfrage an die Wissensdatenbank schaut das System automatisch, ob diese Anfrage aus den aktuellen Fakten & Regeln herleitbar (d.h. beweisbar) ist. Dabei wird für jeden Term in der Anfrage sequentiell durch die Wissensdatenbank geschaut, mit welchen Klauseln dieser gematched werden kann.

Dadurch entsteht der im nachfolgenden Beispiel skizzierte Suchbaum, in welchem mittels Backtracking Lösungen (d.h. ein Beweis) für die Anfrage gesucht werden. alls alle Anfrage-Terme (rekursiv) aufgelöst werden können, liefert das System `true` (ggf. die dazu notwendigen Matchings) zurück, ansonsten `false`.

Bweissuche anhand eines Beispiels:

Wissensdatenbank

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X). % r1
```

Anfrage

```
?- k(Y).
```

Antwort

```
Y=b.
?-
```

Diese Anfrage wird von Prolog folgendermassen abgearbeitet (Suchbaum):

Die *Suchreihenfolge* (in der Wissensdatenbank) ist von oben nach unten. Die Abarbeitungsreihenfolge (Suchbaum, siehe Bild oben) ist von links nach rechts.

2.9 Beispielprobleme

Typische Prolog-Standardprobleme sind:

- Kreuzworträtsel lösen
- Karten färben
- Zahlenrätsel lösen
- Sudoku lösen
- Spracherkennung: Grammatik definieren, korrekte Sätze erkennen, Inhalt „verstehen“ (Semantische Analyse), ...
- Expertensysteme
- u.v.a.m.

Ein paar davon werden hier kurz aufgezeigt.

2.10 Beispiel: Kreuzworträtsel

Ausgangslage:

Gegeben sind 6x4 Felder, LX = Buchstabe an Stelle X:

Erlaubte Wörter (d.h. das „Vokabular“):

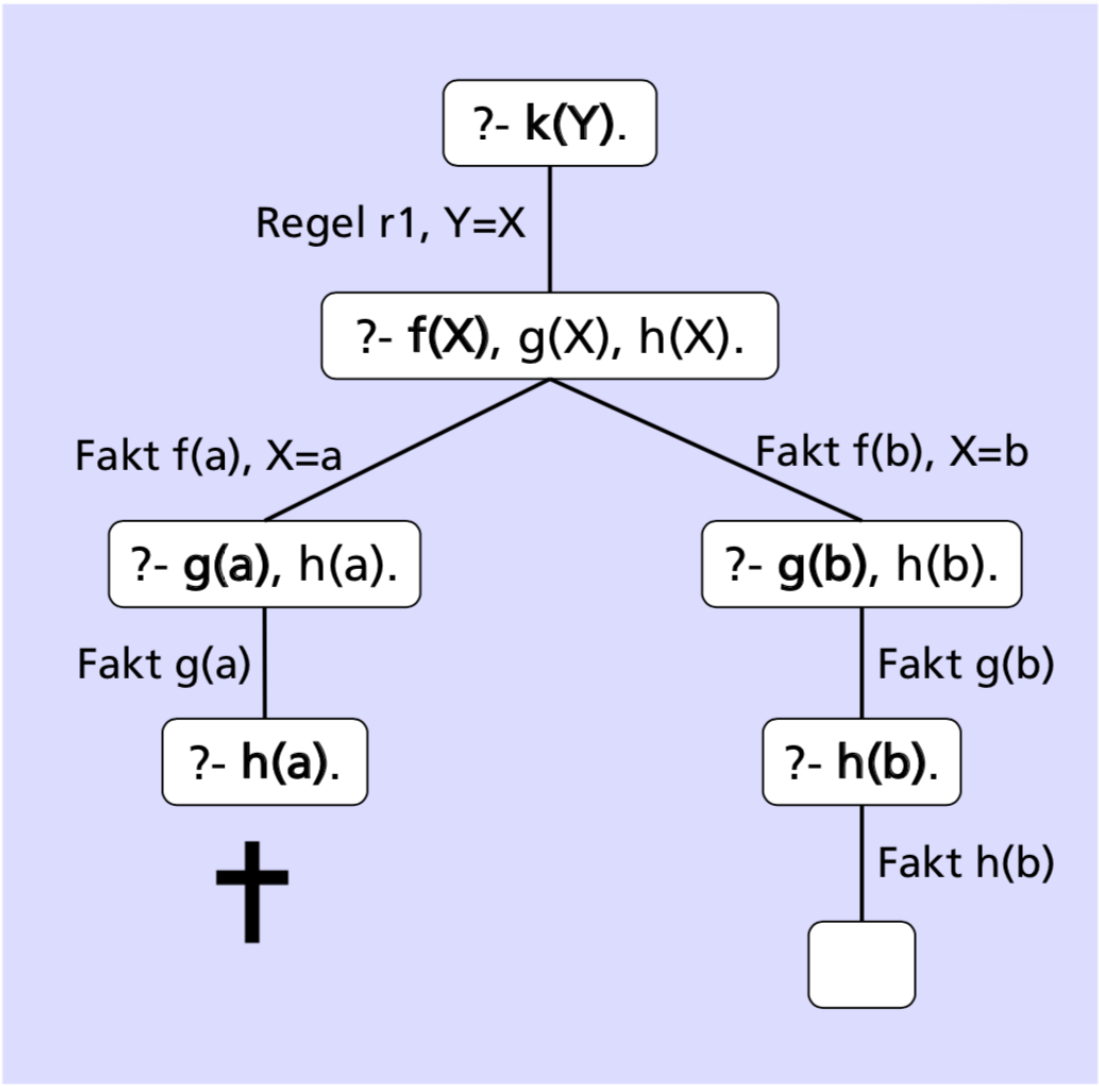

Abbildung 5: Suchbbaum

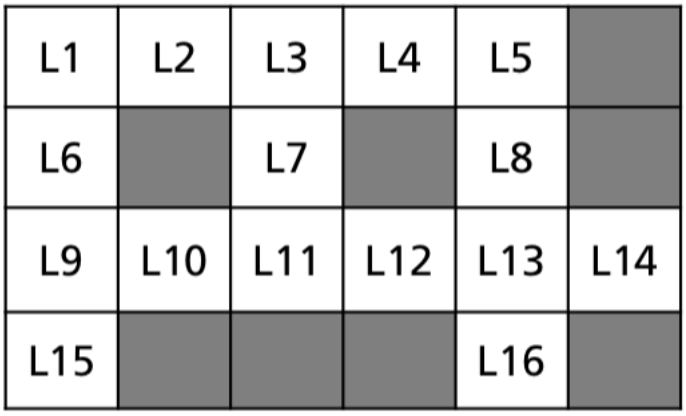

Abbildung 6: Kreuzwort-Rätsel

```
dog, run, top, five,
four, lost, mess, unit,
baker, forum, green, super,
prolog, vanish, wonder, yellow
```

Jede Zelle LX soll mit einem Buchstaben gefüllt werden, so dass in allen zusammenhängenden weissen Zeilen und Spalten erlaubte Wörter stehen.

Vorgehen:

Alle erlaubten Wörter erfassen, und zwar als Prädikat `word/n`. Zum Beispiel:

```
word(d, o, g).
```

dog wird hier als ein gültiges Wort mit drei Buchstaben definiert. Ausserdem müssen wir Regeln für alle Zeichen LX definieren. Darin soll beschrieben sein, dass wir 16 Zeichen suchen, und dass sie zusammen ein erlaubtes Wort bilden müssen.

```
% words that may be used in the solution

word(d,o,g). word(r,u,n). word(t,o,p).
word(f,i,v,e). word(f,o,u,r). word(l,o,s,t). word(m,e,s,s). word(u,n,i,t).
word(b,a,k,e,r). word(f,o,r,u,m). word(g,r,e,e,n). word(s,u,p,e,r).
word(p,r,o,l,o,g). word(v,a,n,i,s,h). word(w,o,n,d,e,r). word(y,e,l,l,o,w).

solution(L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16) :-
word(L1,L2,L3,L4,L5), % Top horizontal word
word(L9,L10,L11,L12,L13,L14), % Second horizontal word
word(L1,L6,L9,L15), % First vertical word
```

```
word(L3,L7,L11), % Second vertical word
word(L5,L8,L13,L16). % Third vertical word
```

Lösung:

Damit sind wir fertig, das Problem ist dadurch beschrieben. Aufgerufen wird die Lösung so:

```
?- solution(L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,L15,L16).
L1 = f,
L2 = o,
L3 = r,
L4 = L7, L7 = u,
L5 = m,
L6 = L12, L12 = i,
L8 = L15, L15 = e,
L9 = v,
L10 = a,
L11 = n,
L13 = L16, L16 = s,
L14 = h
false.
```

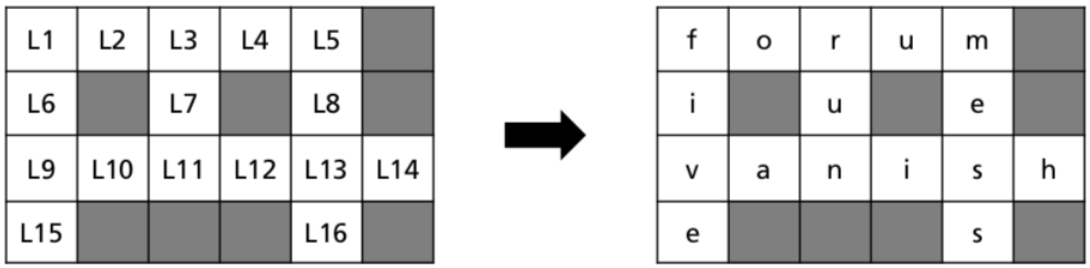

Abbildung 7: Lösung Kreuzworträtsel

2.11 Beispiel: Karten färben

Ausgangslage:

Eine Karte mit verschiedenen Ländern und 3 verschiedenen Farben, so dass jedes Land eine Farbe hat, und benachbarte Länder jeweils unterschiedliche Farben haben.

Vorgehen:

Neues Prädikat n/2 (für „Nachbar“):

Abbildung 8: Map

n(red, green).

Das bedeutet für uns, dass rot und grün nebeneinander sein können. Welche Länder nebeneinander sind, modellieren wir mit dem selben Prädikat:

```
n(CH, I).
```

Die Schweiz ist ein Nachbar von Italien. (Die Länder sind mit Grossbuchstaben geschrieben und somit Variablen. Diesen soll dann eine Farbe zugewiesen werden.)

Alle definierten `n(X,Y)` Regeln müssen natürlich gleichzeitig erfüllt sein. Deshalb fügen wir ein weiteres Prädikat `colors/7` ein, dass die Nachbarschaften von den Ländern regelt.

Um die Anzahl Lösungen klein zu halten, fügen wir eine zusätzliche Einschränkung ein:

```
CH = red
```

Das ganze Programm sieht dann so aus:

```
% possible pairs of color of neighboring countries
n(red, green). n(red, blue).
n(green, red). n(green, blue).
n(blue, red). n(blue, green).

% countries and neighbors
colors(CH, A, D, I, F, B, N) :-
CH = red,
n(CH, A), n(CH, I), n(CH, F), n(CH, D),
n(A, D), n(A, I),
n(I, F),
n(F, B),
n(D, B), n(D, N),
n(B, N).

% colors(CH, A, D, I, F, B, N).
```

Lösung:

```
?- colors(CH, A, D, I, F, B, N).
CH = B, B = red,
A = F, F = N, N = green,
D = I, I = blue
```

Der fehlende Punkt am Ende zeigt uns an, dass es noch weitere Lösungen gibt. Diese können mit `;` angezeigt werden. Danach sieht die Ausgabe so aus:

```
?- colors(CH, A, D, I, F, B, N).
CH = B, B = red,
A = F, F = N, N = green,
D = I, I = blue
CH = B, B = red,
A = F, F = N, N = blue,
D = I, I = green
false.
```

2.12 Operatoren

Wenn man Prolog fragt, was `X = 1 + 2` ist, bekommt man folgende Antwort:

```
?- X = 1 + 2.
X = 1+2.
```

Der Operator `=/2` macht nur ein Matching, Ausdrücke werde nicht automatisch ausgewertet. Dafür kann der `is/2` Operator genutzt werden.

```
?- X is 1 + 2.
X = 3.
```

Es gibt zahlreiche eingebaute arithmetische Operatoren:

- `+`: Addition
- `-`: Subtraktion
- `*`: Multiplikation
- `/`: Division
- `**`: Potenz
- `//`: Ganzzahldivision
- `mod`: Modulo

Weitere: http://www.swi-prolog.org/pldoc/man?section=functions

Einige Beispiele:

```
?- D is 5/2.
D = 2.5.

?- I is 5//2.
I = 2.
```

```
?- C is cos(0).
C = 1.0.

?- S is sqrt(9).
S = 3.0.
```

Vordefinierte arithmetische Vergleichsoperatoren:

- >: grösser als
- <: kleiner als
- >=: grösser gleich
- =<: kleiner gleich
- =:=: Gleichheit
- =\=: Ungleichheit

Hinweis: Diese Operatoren erzwingen die arithmetische Auswertung ihrer beiden Operanden.

SWI-Doku: http://www.swi-prolog.org/pldoc/man?section=arithpreds

Beispiele:

```
?- 88 =< 77.
false.

?- 44 =\= 42 + 1.
true.

?- 1 + 2 = 2 + 1.    % '=' is for matching!
false.               % '1+2' does not match '2+1'

?- 1 + 2 =:= 2 + 1.
true.
```

2.12.1 Operatoren und Prädikate

Grundsätzlich gibt es in Prolog nur Prädikate. Operatoren sind lediglich Prädikate mit einer anderen Schreibweise.

Beispiel =/2:

```
?- =(tom, tom).      % Prädikatenschreibweise
true.

?- tom = tom.        % Operatorenschreibweise
true .
```

Mit dem eingebauten Prädikat `op/3` können Prädikate als Operatoren deklariert werden:

```
?- op(1150, xfx, is_bigger).    % declare new operation
true.

?- elephant is_bigger dog.      % use our new operation
true.
```

Ein paar Beispiele bereits vordefinierter Operatoren:

Präzedenz	Typ	Name
1200	`xfx`	`-->`, `:-`
1200	`fx`	`:-`, `?-`
1100	`xfy`	`;`, `|`
1000	`xfy`	`,`
900	`fy`	`\+`
700	`xfx`	`<`, `=`, `=:=`, `=\=`, `is`, ...
500	`yfx`	`+`, `-`, `/\`, `\/`, `xor`
400	`yfx`	`*`, `/`, `//`, `rdiv`, `<<`, `>>`, `mod`, `rem`
200	`xfx`	`**`
200	`fy`	`+`, `-`, `\`

Ganze Tabelle: http://www.swi-prolog.org/pldoc/man?predicate=op/3

Die **Präzedenz** (oder Operatorrangfolge) von einem Operator gibt an, wie stark dieser Operator seine Operanden bindet. Je tiefer die Präzedenz, desto stärker ist die Bindung.

Der **Typ** gibt die relative Reihenfolge von Operator `f` und den Operanden `x`, `y` an. Es gibt drei Gruppen:

- Infix: `xfx`, `xfy`, `yfx`
- Präfix: `fx`, `fy`
- Postfix: `xf`, `yf`

Mit `x` und `y` wird die Präzedenz der Operatoren einer Operation festgelegt:

- `x` repräsentiert einen Operanden, dessen Präzedenz strikt kleiner ist als die Präzedenz vom Operator `f`.
- `y` repräsentiert einen Operanden, dessen Präzedenz kleiner oder gleich derjenigen vom Operator `f` ist.

Damit wird festgelegt, wie Ausdrücke mit mehreren gleichen Operatoren ausgewertet werden, z.B. bei `a - b - c` oder `1000 / 100 / 10`.

Wenn man mehrere Operatoren vom Typ `xfx` nacheinander schreibt, ist also nicht klar geregelt, in welcher Reihenfolge der Ausdruck ausgewertet werden soll, und es gibt eine Fehlermeldung:

```
?- X is Y is 42.
ERROR: Syntax error: Operator priority clash
ERROR: X i
ERROR: ** here **
ERROR: s Y is 42 .
```

2.13 Rekursion

Prädikate in Prolog können rekursiv definiert sein, d.h. eine oder mehrere Regeln beziehen sich in ihrer Definition auf sich selber. Dies wird in Prolog sehr häufig verwendet.

Im Prinzip wird Rekursion immer gleich eingesetzt, ein Problem wird in Fälle aufgeteilt, welche zu einer der folgenden beiden Gruppen gehören:

1. Einfache Fälle oder Grenzfälle, „direkt“ lösbar
2. Allgemeine Fälle, zu denen die Lösung mithilfe von Lösungen von (einfacheren) Versionen vom gleichen Problem konstruiert werden kann

Beispiel `is_bigger`:

```
is_bigger(X, Y) :- bigger(X, Y).                    % simple case
is_bigger(X, Y) :- bigger(X, Z), is_bigger(Z, Y).   % general case
```

Beispiel Fakultät:

```
fak(0, 1).          % simple case
fak(N, F) :-        % general case
    N > 0,          %   argument test
    N1 is N-1,      %   evaluateN-1
    fak(N1, F1),    %   recursive call
   F is N * F1.     %   sum up
```

Query:

```
?- fak(5, X).
X = 120
false.
```

Beispiel Fibonacci:

```
fib(0, 0).
fib(1, 1).
fib(N, F) :-
    N > 1,
    N1 is N - 1,
    N2 is N - 2,
    fib(N1, F1),
    fib(N2, F2),
    F is F1 + F2.
```

Query (7. Fibonacci-Zahl):

```
?- fib(7, X).
X = 13 .
```