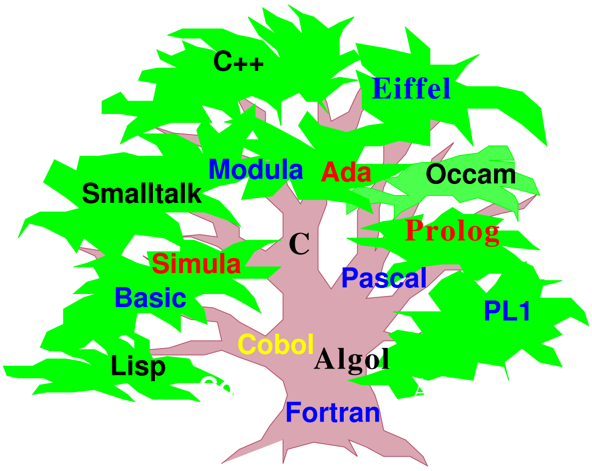
7. Funktionale und logische Programmierung

- Grundlagen funktionaler Programmierung
- Scheme: Ein LISP-Dialekt
- Grundlagen logischer Programmierung
- Prolog



Funktionale Programmierung

- Hauptunterschiede zu imperativer Programmierung
  - keine Variablen
  - keine Zuweisungen
  - keine Schleifen (sondern Rekursion)
- Anwendung von Funktionen im mathematischen Sinne
  - Abbildung zwischen Werten eines Definitions- in Werte eines Wertebereiches

  f: D -> W   z.B. f (x) = 10 * x

- Prozeduren imperativer Programmiersprachen leiden unter Seiteneffekten (Zuweisungen an globale Variablen bzw. Referenzparameter)

```
PROCEDURE strange (): INTEGER;
BEGIN
   globalesA := globalesA + 1;
   RETURN globalesA;
END strange;
```

- Funktionale Programmierung umgeht Seiteneffekte und gewährleistet *referentielle Transparenz*
  - Funktionsanwendung auf einem bestimmten Argument liefert immer denselben Ergebniswert

Funktionale Programmierung (2)

- Haupbestandteile funktionaler Programmiersprachen
  - Menge primitiver Funktionen
  - Mechanismus zur Definition neuer aus bestehenden Funktionen
  - Anwendung von Funktionen innerhalb von Ausdrücken

- auch mit imperativen Programmiersprachen kann funktional programmiert werden

- Beispiel: Funktionales Programm, um Differenz zwischen der größten und kleinsten von drei Zahlen zu ermitteln

```
PROCEDURE max (a, b: INTEGER): INTEGER;
BEGIN IF a > b THEN RETURN a ELSE RETURN b END
END max;

PROCEDURE min (a, b: INTEGER): INTEGER;
BEGIN IF a < b THEN RETURN a ELSE RETURN b END
END max;

PROCEDURE diff (a, b, c: INTEGER): INTEGER;
BEGIN
       RETURN
END diff;
```


Funktionale Programmierung (3)

- Reihenfolge der Funktionsauswertung ist nicht festgelegt
  - bessere Eignung zur Parallelverarbeitung als mit imperativer Programmierung

- zentrale Datenstruktur funktionaler Sprachen: Liste
  - Elemente: Atome (Namen, Konstanten) bzw. andere Listen
  - vordefinierte Funktionen wie **head, remainder** und **cons** (Konstruktion)

```
[A B C]
[[A B] [Hello there] 43]
```

- Verarbeitung von Listenstrukturen führt stets auf 3 Fälle
  - leere Liste (nil)
  - Verarbeitung des ersten Elements (head)
  - rekursive Abarbeitung des Rests (remainder)

```
PROCEDURE isMember (x: INTEGER; a: INTEGER_Liste): BOOLEAN;
BEGIN
    IF a = NIL THEN RETURN FALSE
           ELSIF x = head (a) THEN RETURN TRUE
           ELSE RETURN isMember (x, remainder (a))
    END
END isMember;
```

Funktionale Programmierung (4)

- ***Funktionen höherer Ordnung:*** Argumente und Ergebnis können auch Funktionen sein

- Beispiele:
  - Komposition o

    f: X -> Y; g: Y -> Z
    g o f: X -> Z   mit   (g o f) (x) = g (f (x))

  - Konstruktion K: Anwendung einer Liste von Funktionen auf ein Argument

    K ([f g h], a) = [f (a) g (a) h (a)]

  - Funktionsanwendung auf alle Elemente einer Liste: *map*

    map (f, [l1, l2, l3]) = [f(l1) f(l2) f (l3)]

- Realisierung für map

```
PROCEDURE map (f: PROCEDURE, L: Liste): Liste;
BEGIN
   IF L=NIL THEN RETURN NIL
   ELSE RETURN cons ( f(head (L)), map (f, remainder (x)))
   END
END map;
```


Scheme: Ein LISP-Dialekt

- LISP
  - Entwicklung am MIT Ende der 50er / Anfang der 60er Jahre
  - mit seinen Dialekten am weitesten verbreitete funktionale Programmiersprache
  - aufgrund imperativer Sprachelemente (Zuweisung) keine rein funktionale Sprache!
  - modernere Varianten: Scheme (statische Gültigkeitsbereiche) und CommonLISP

- intensive Verwendung von S-Ausdrücken
  - Atome: Zahlen, Zeichenketten, Namen, vordefinierte Funktionen
  - Listen: `( )`, `( a )`, `(+ 5 6)`

- Ausdrücke sind in Präfixform anzugeben

```
(+ 4 (* 5 3))
(and (= a b) (<> a 0))
(ggt 10 35)
(read)
```

- Definition neuer Funktionen durch spezielle Lambda-Ausdrücke

```
(lambda Parameterliste Funktionsrumpf)
```

```
z.B.          (lambda (x) (* x x))
```

Scheme (2)

- Definition benannter Funktionen

```
(define sqr (lambda (x) (* x x))
```

ist äquivalent zu

```
(define (sqr x) (* x x))
```

- Vordefinierte Funktionen auf S-Ausdrücken
  - **car** liefert erstes Element
  - **cdr** liefert Rest einer Liste (S-Ausdrucks)
  - **cons** Konstruktion eines neuen Ausdrucks aus zwei Teilen
  - null? Test auf leere Liste
  - atom? Test auf Atom
  - number? Test auf Zahl

- Vewendung eines S-Ausdrucks als reine Datenfolge: **quote** bzw. **’**

```
(car (quote (A B C)))
```


Scheme (3)

- bedingte Auswertung: (**if** <Ausdruck1> <Ausdruck2> <Ausdruck3>)

```
if (= a 0)
      0
      (/ 1 a)
```

- Verallgemeinerung: (**cond** (<Ausdr*11*> <Ausdr*12*>)...(<Ausdr*n1*> <Ausdr*n2*>))
  - Auswertung der <Ausdruck*i1*> bis TRUE: Ergebnis von <Ausdruck*i2*> wird zurückgeliefert
  - sind alle Bedingungen nicht erfüllt, wird nil als Ergebnis geliefert

```
cond        ( (= a 0) 0)
            ( (= a 1)  1)
            ( T (/ 1 a))
```

- Beispiel: Funktion *isMember*

```
(define (isMember x L)
    (cond      ( (null? L) nil )
               ( (= x (car L)) T)
               ( T (isMember x (cdr L)))
    )
)
```

Scheme (4)

- Realisierung höherer Funktionen

```
(define (compose g f)
   (lambda (x) (g (f x))))


(define (map f L)
         (if (null? L) ()
                (cons ( f (car L)) (map f (cdr L)))))

(define (sqr-lis L) (map sqr L)
```

- funktionale Sprachen verwenden automatische Speicherzuteilung und -freigabe
- neuere funktionale Sprachen mit statischer Typisierung: ML und Miranda

Logische Programmierung

- Logik: Wissenschaft vom Schließen und Beweisen
- Logische Programmierung: Verwendung logischer Aussagen zur Durchführung von Berechnungen
- formale Darstellung logischer Aussagen: Prädikatenkalkül 1. Ordnung

umgangssprachliche Aussagen:	***Aussagen im Prädikatenkalkül:***
0 ist eine natürliche Zahl.	natürlich (0).
2 ist eine natürliche Zahl.	natürlich (2).
für alle x, wenn x eine natürliche Zahl ist, dann ist der Nachfolger von x auch eine natürliche Zahl.	für alle x, natürlich (x) -> natürlich (nachfolger (x)).
-1 ist eine natürliche Zahl.	natürlich (-1).

- Axiome: Aussagen, die als wahr angenommen werden
  - Axiome dienen zum Beweis anderer wahrer Aussagen

Logische Programmierung (2)

- Bestandteile von Aussagen im Prädikatenkalkül erster Ordnung
  - Konstanten (Atome): Zahlen, Namen
  - Prädikate: Boolesche Funktionen
  - (allg.) Funktionen
  - Variablen: Platzhalter für noch nicht spezifizierte Größen
  - Junktoren: Boolesche Operatoren (oder, und, nicht), Implikation (->), Äquivalenz (<->)
  - Quantoren: Allquantor ("für alle") und Existenzquantor ("es ex. mind. ein")
  - Interpunktionssymbole: **( ) , .**

- durch Quantor eingeführte Variablen heißen ***gebunden***, sonst ***frei***

- als Argumente von Prädikaten und Funktionen sind nur ***Terme*** zugelassen: Kombinationen von Variablen, Konstanten und Funktionen (keine Prädikate, Quantoren oder Junktoren)

- Ableitungsregeln (Inferenzregeln): bestimmen, wie neue aus vorgegebenen Aussagen abzuleiten oder zu beweisen sind, z.B.

$$\frac{(a \rightarrow b) \text{ und } (b \rightarrow c)}{a \rightarrow c}$$

Logische Programmierung (3)

- logische Programme legen nur die Menge von Axiomen fest, nicht wie Ableitung anderer Aussagen damit zu erfolgen hat (deklarative Programmierung)

- Eingabe für die Programme sind Anfragen (queries, goals), für die Wahrheitswert zu bestimmen ist

- Beispielprogramm

```
säugetier (pferd).
säugetier (mensch).
für alle x, säugetier (x) ->    (beine (x,4) und arme (x,0)) oder
                                ((beine (x,2) und arme(x,2)).
arme (pferd,0).
nicht arme (mensch,0).
```

- Anfragen:

```
arme (mensch, 2)?

beine (mensch, 0)?

gibt es ein y, so daß beine (mensch, y)?
```

Prolog (Programming in Logic)

- Entwicklung in den 70er Jahren
  - zunächst Univ. Marseille (Colmerauer, Roussel), später auch an Univ. Edinburgh (Kowalski)
  - Verbreitung v.a. seit den 80er Jahren (Japan: 5th Generation Project)
  - keine Standardisierung

- Prolog-Notation für Aussagen
  - Prolog-Anweisung: setzt sich zusammen aus atomaren Formeln
  - Atomare Formel:

$$\underbrace{p}_{\text{Prädikats-symbol}} \; \underbrace{(U, V, ..., Z)}_{\text{Argumentliste}} \qquad \text{n-stelliges Prädikat}$$

  - Argumente: Konstanten, Variable, Funktionssymbole
  - Prädikatssymbol: Boole-wertige Funktion, die auf die Argumentliste angewendet wird

- Konventionen
  - Prädikatssymbole, Funktionssymbole und Konstanten beginnen mit einem Kleinbuchstaben (Konstante können auch Integer sein)
  - Variable müssen mit einem Großbuchstaben beginnen

- keine Typisierung!

Prolog (2)

- Darstellung von Aussagen in Form sogenannter *Horn-Klauseln*

$$\underbrace{B}_{\text{Konsequenz (Kopf, Goal)}} \; :- \; \underbrace{A_1, A_2, ..., A_n}_{\text{Voraussetzung (Antezedenz), Rumpf}} .$$

  - Bedeutung: wenn Aussagen A_1 und A_2 und ... A_n wahr ist, dann ist B wahr
  - äquivalent zu Axiom: B or not (A_1) or not (A_2) or ... or not (A_n)

- Konventionen:
  - Variablen im Kopf sind durch den Allquantor gebunden
  - Variablen, die nur im Rumpf (nicht im Kopf) auftreten, sind durch den Existenzquantor gebunden

- geringere Ausdrucksmächtigkeit als mit Prädikatenlogik

- es sind drei Arten von Klauseln zu unterscheiden
  - ***Regeln***: rechte und linke Seite vorhanden
  - ***Fakten*** (Axiome): leere rechte Seite (n=0) -> Aussagen, die ohne Voraussetzung erfüllt sind
  - ***Anfragen*** : leere linke Seite

Prolog (3)

- Beispielprogramm

Fakten:

```
säugetier (pferd).                säugetier (mensch).
arme (pferd,0).                   not arme (mensch,0).
```

Regel: für alle x, säugetier (x) -> (beine (x,4) und arme (x,0)) oder ((beine (x,2) und arme(x,2)).

Realisierungsmöglichkeit in Prolog:

- Bearbeitung von Anfragen
  - das System versucht, eine Ableitung durch Anwendung “passender” Regeln vorzunehmen (bzw. eine Übereinstimmung zu bestehenden Fakten zu ermitteln)

```
? beine (mensch, 2)                ? beine (pferd, 2)
```


Prolog (4)

- bei den Ableitungen ist ein “Pattern Matching” vorzunehmen: ***Unifikation***
  - in Anfragen vorgegebene Werte sind auf Variablen in den Regeln abzubilden
  - Variablen in den Anfragen sind auf Variablen oder Konstanten in den Regeln abzubilden
- die Berechnung erstreckt sich i.d.R. über zahlreiche Ableitungsschritte, wobei in den meisten Schritten mehrere Regeln anwendbar sind -> große Suchräume
- falls bestimmter Ableitungspfad zu keinem Ergebnis führt, wird versucht, die Berechnung an einen Punkt zurückzuführen, an dem weitere Regel anwendbar sind (***Backtracking***)
- Beispiel:

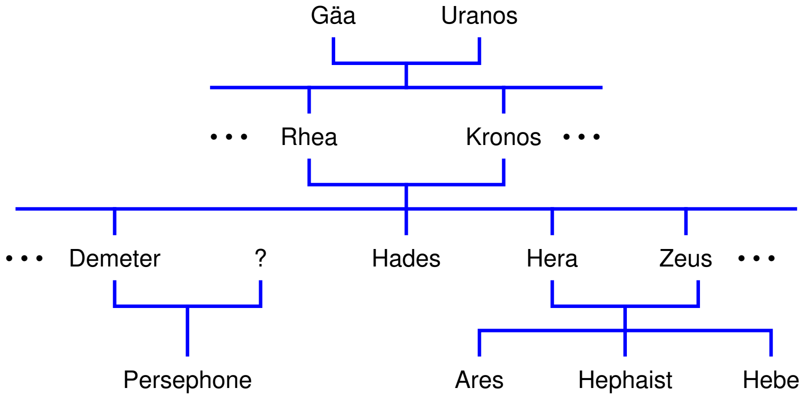
```
? beine (pferd, Y)
```

Weiteres Beispiel

Logik-Programm

Fakten

geschlecht (gäa, w).
geschlecht (uranos, m).
geschlecht (rhea, w).
geschlecht (kronos, m).
•••
mutter (gäa, rhea).
vater (uranos, rhea).
mutter (gäa, kronos).
vater (uranos, kronos).
•••



Regeln

(1) elternteil (M, K)	:–	mutter (M, K).
(2) elternteil (V, K)	:-	vater (V, K).
(3) geschwister (X, Y)	:-	elternteil (A, X), elternteil (A,Y), X ≠ Y.
(4) tante (T, K)	:-	elternteil (X, K), geschwister (T,X), geschlecht (T,w).
(5) onkel (O, K)	:–	elternteil (X, K), geschwister (O,X), geschlecht (O,m).
(6) cousin (X, Y)	:–	elternteil (A, X), elternteil (B,Y), geschwister (A,B).
(7) vorfahr (X, Y)	:–	elternteil (X, Y).
(8) vorfahr (X, Y)	:–	elternteil (X, Z), vorfahr (Z, Y).

Rekursive Regeln

- Regeln, bei denen Kopfprädikat im Regelrumpf auftritt
- wichtige Klasse rekursiver Probleme: Berechnung der transitive Hülle
  - Die ***transitive Hülle*** einer binären Relation r ist die kleinste Relation s, die r einschließt und transitiv ist, d.h. s (X,Y), s (Y,Z) impliziert s (X,Z)

Beispiel: vorfahr-Prädikat als transitive Hülle des elternteil-Prädikats

vorfahr (X, Y) :- elternteil (X, Y).
vorfahr (X, Y) :– elternteil (X, Z), vorfahr (Z, Y).

- Variationen:
  - lineare Rekursion: Kopfprädikat tritt in Regelrumpf (Antezedenz) genau einmal auf
  - nicht-lineare Rekursion: rekursives Prädikat tritt mehrfach in Antezedenz auf
  - manchmal Unterscheidung nach links-, rechts- oder mittel-rekursiv bei linearen Problemen

	closure (X, Y) :- basis (X, Y).
links-rek.:	**closure (X, Y) :- closure (X, Z), basis (Z, Y).**
rechts-rek.:	**closure (X, Y) :- basis (X, Z), closure (Z, Y).**

Suchstrategie

- Zur Ableitung von Zielen bestehen zwei generelle Vorgehensweisen:
  - ***Forward chaining*** (bottom-up resolution): ausgehend von den Fakten wird versucht, durch Regelanwendung das Ziel herzuleiten
  - ***Backward chaining*** (top-down resolution): ausgehend vom Ziel wird versucht, durch Regelanwendung eine Abbildung auf die Fakten zu erreichen

- die Ableitung mehrerer Teilziele kann auf zwei Arten erfolgen
  - Tiefensuche (depth-first): die Teilziele werden nacheinander vollständig “abgearbeitet”
  - Breitensuche (breadth-first): alle Teilziele werden parallel abgeleitet

- Prolog: Backward Chaining mit Tiefensuche unter Verwendung von Backtracking
  - streng lineare Strategie zur Abarbeitung von Regeln
  - Regeln werden in der Reihenfolge, wie sie im Programm angegeben werden (von oben nach unten), abgearbeitet
  - Teilziele im Rumpf einer Regel werden von links nach rechts abgearbeitet
  - Berücksichtigung weiterer Teilziele erfolgt erst, wenn Teilziel vollständig abgearbeitet wurde

- links-rekursive Regeln können in Prolog zu Endlosschleifen führen !

Suchstrategie (2)

- Reihenfolge der Prädikate wesentlich für Effizienz

  ? geschlecht (X, m), geschwister (hera, X)

  - selektivste Prädikate sollten zuerst angegeben werden

- Beschneidung des Suchraums durch CUT-Operator
  - Cut-Operator (!) bewirkt, daß danach folgende Prädikate und Regeln nicht mehr abgearbeitet werden
  - größere Effizienz, jedoch Gefahr von fehlerhaften (unvollständigen) Ergebnissen

```
R :- G1, G2, !, G3.
R :- G4.
```

Prolog-Lösung für: IF x < 0 THEN F := -1 ELSIF x = 0 THEN F := 0 ELSE F := 1 END;

```
f (X, -1)  :-   X < 0.
f (X, 0)   :-   X = 0.
f (X, 1)   :-   X > 0.
```

- Prolog-Programmierer muß Suchstrategie des Systems genau kennen und ist in hohem Maße für Effizienz der Verarbeitung verantwortlich

Negation (CWA)

- was nicht als wahr bewiesen werden kann, wird als falsch angesehen (Closed World Assumption)
- aufgrund unvollständiger Fakten werden oft falsche Aussagen geliefert

```
? vater (X, persephone)
```

- Anfrage `not (X)` ist erfolgreich, wenn das Ziel `X` nicht abgeleitet werden kann (Interpretation der Negation als Mißerfolg)

```
? not vater (X, persephone)
```

- `not (not (X))` liefert nicht unbedingt dasselbe Ergebnis wie `X`

```
männlich (uranos).
? männlich (X).
? not (not (männlich (X)).
```

- Position negierter Prädikate innerhalb einer Anfrage oder Regel beeinflußt Ergebnis

```
? X=0, not (X=1).

? not (X=1), X=0.
```


Zusammenfassung

- Funktionale und logische Sprachen
  - höheres Abstraktionsniveau als imperative Programmiersprachen
  - weitgehend deklarativer Programmierstil ist möglich
  - fehlende/dynamische Typisierung: große Flexibilität, jedoch hohe Ausführungskosten
- (rein) funktionale Sprachen
  - referentielle Transparenz durch Verzicht auf Variablen und Zuweisungen
  - geschachtelte und rekursive Funktionsbearbeitung auf Listenstrukturen
  - hohe Ausdrucksmächtigkeit durch Nutzung höherer Funktionen
- Logische Programmierung
  - Berechnungen erfolgen durch Ableitung von Aussagen (Regelanwendung auf Fakten)
  - Prolog: Verwendung von Horn-Klauseln und eingebauter Inferenzstrategie
  - Probleme: einfache Suchstrategie, Programmierer für Effizienz sowie korrekte Ableitung mitverantwortlich, Behandlung der Negation
- Anwendungsbereiche
  - Künstliche Intelligenz
  - mathematische Beweissysteme
  - Logikanwendungen (Expertensysteme, deduktive Datenbanken, ...)