5 Logische Programmierung

- Logik wird als Programmiersprache benutzt
- Der logische Ansatz zu Programmierung ist (sowie der funktionale) deklarativ; Programme können mit Hilfe zweier abstrakten, maschinen-unabhängigen Konzepte verstanden werden:
  - ▷ Wahrheit
  - ▷ Logische Deduktion

Deklarativität der logischen Programmierung

- Das auszuführende Program wird spezifiziert durch
  - ▷ das Wissen über ein Problem und die Annahmen, die hinreichend für die Lösung sind ≡ logische Axiomen;
  - ▷ eine zu beweisende Aussage (Ziel, goal statement) als das zu lösende Problem. (≈ Eingabe)
- Die Ausführung
  - ▷ ist der Versuch das goal statement zu beweisen unter den gegebenen Annahmen.

5.1 Grundlegende Konstrukte

Die Hauptkonstrukte der logischen Programmierung stammen aus der Logik:

- Aussagen: Fakten, Anfragen und Regeln
- Terme $\equiv$ die einzigen Datenstrukturen

Fakten

- Fakten sind Aussagen die Beziehungen zwischen Objekten definieren

  ```
  father(abraham,isaac).
  ```

  ... besagt, dass zwischen `abraham` und `isaac` die Beziehung `father` besteht.

- Eine Beziehung kann man als ein Prädikat auffassen: das Prädikat `father` gilt für `abraham` und `isaac`.

Fakten

- Die *Plus*-Beziehung:

  plus(0,0,0). plus(0,1,1). plus(0,2,2). plus(0,3,3).

  plus(1,0,1). plus(1,1,2). plus(1,2,3). plus(1,3,4).

  plus(2,0,2). plus(2,1,3). plus(2,2,4). plus(2,3,5).

- Eine endliche Menge von Fakten ist das einfachste Programm.

Anfragen

Anfragen (*queries*) erlauben es zu testen, ob eine Beziehung zwischen Objekten besteht, und somit Informationen aus einem Programm abzufragen:

```
father(abraham,isaac)?
```

... fragt, ob die Beziehung `father` zwischen `abraham` und `isaac` besteht.

Anfragen

- Um eine Anfrage für ein gegebenes Programm zu beantworten, muss man bestimmen, ob die Anfrage eine logische Folge des Programms (d.h. der spezifizierten Axiomen) laut der Deduktionsregeln ist.

- Die einfachste Deduktionsregel ist Identität:

  aus P folgt P,

  d.h. eine Frage ist die logische Folge eines identischen Fakten.

  $\Longrightarrow$ `father(abraham,isaac)?` ist wahr, angenommen der Fakt `father(abraham,isaac).` ist Teil des Programms.

Logische Variablen

Statt nur `wahr` oder `falsch` als Antworten zu bekommen, möchte man manchmal auch Objekte mit einer bestimmten Eigenschaft herausfinden. Z.B.:

Wessen Vater ist Abraham?

- 1. Möglichkeit Wiederholte Anfragen:

  `father(abraham,lot)?`, `father(abraham,milcah)?`, ...,

  `father(abraham,isaac)?`, ...

  bis man die Antwort `wahr` erhält.

- 2. Möglichkeit Besser, Variablen benutzen, um über unbekannte Objekte zu sprechen...

Terme

- Die Objekte eines Programms werden als Terme spezifiziert. Induktive Definition:
  - ▷ Atome: Konstanten (z.B. `abraham`, `lot`, `0`, `1`) sind Terme.
  - ▷ Variable: Variable (z.B `X`, `Y`) sind Terme.
  - ▷ Zusamengesetzte Terme: Wenn t_1, t_2, $\ldots$, t_n Terme sind, und f ein Name ist, dann ist $f(t_1, t_2, \ldots, t_n)$ ein Term.
    - f heißt Funktor, t_1 bis t_n Argumente; n ist die Stelligkeit des Funktors
- Syntaktische Konvention: Nur Variablennamen werden großgeschrieben.

Terme

- Beispiele:
  - `name(john,smith)`
  - `name(X,Y)`
  - `person(name(john,smith),age(23))`
  - `person(name(X,smith),age(23))`
  - `person(Y,age(23))`
  - `node(node(leaf(a),leaf(b)),leaf(c))`
  - `s(0)`
- Terme sind die einzige Datenstruktur in logischen Programmen.

Substitutionen und Instanzen

- Eine Substitution ist eine endliche Menge von Paaren der Form $X_i = t_i$, mit X_i eine Variable, t_i ein Term und $X_i \neq X_j$ für alle $i \neq j$ und X_i tritt in keinem t_j auf für alle i und j.
  - ▷ Bsp.: {`X=isaac`}
- Die Anwendung einer Substitution θ auf einen Term A, $A\ \theta$, ist der Term, den man erhält, indem man jedes Auftreten von X durch t ersetzt für jedes Paar $X = t$ in θ.
- A ist eine Instanz von B, wenn eine Substitution θ existiert, so dass $A = B\ \theta$.
  - ▷ Bsp.: `father(abraham,isaac)` ist eine Instanz von `father(abraham,X)` unter der Substitution {`X=isaac`}.

Variablen in Anfragen

▷ `father(abraham,X)?`

• Anfragen mit Variablen haben eine existentielle Interpretation:

Existiert eine Substitution, für welche die Anfrage eine logische Folge des Programms ist?

$\Longrightarrow$ Verallgemeinerung als Deduktionsregel:
Eine existentielle Anfrage $P?$ ist eine Folge einer Instanz $P\ \theta$ für jedes θ.

▷ `father(abraham,X)` ist eine Folge von `father(abraham,isaac).`

Variablen in Anfragen

`father(abraham,X)?`

- Der Beweis einer Anfrage ist konstruktiv, d.h. falls die Anfrage mit *ja* beantwortet werden kann, wird eine Substitution ausgegeben, für die die Aussage aus dem Programm deduzierbar ist.

  Antwort: `yes,` $\{$`X`$\mapsto$ `isaac`$\}$

Variablen in Fakten

- Variablen in Fakten sind universal quantifiziert. Ein Fakt $p(T_1, T_2, \ldots, T_n)$ besagt, dass für jedes $X_1, \ldots, X_k$, mit X_i eine Variable die im Fakt auftritt, $p(T_1, \ldots, T_n)$ wahr ist.
  - ▷ Bsp.: `likes(X,pomegranates)` besagt, dass für alle `X`, X Granatäpfel mag.

$\Longrightarrow$ Instanziierung als Deduktionsregel: Aus einer universell quantifizierten Aussage P folgt eine Instanz $P\ \theta$ für jede Substitution θ.

- ▷ Bsp.: `likes(lot,pomegranates)` folgt aus dem Fakt `likes(X,pomegrantes).`

Konjunktive Anfragen

- Eine Aussage $p(t_1, t_2, \ldots, t_n)$ heißt auch Ziel (engl. *goal*).
- Eine Anfrage $p(t_1, t_2, \ldots, t_n)?$, die aus nur einem Ziel besteht heißt einfach.
- Eine Anfrage kann auch eine Konjunktion aus mehreren Zielen sein $\Longrightarrow$ konjunktive Anfragen.
- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist.
- Bsp. `father(abraham,isaac),male(lot)?`

Konjunktive Anfragen mit Variablen

- Wenn eine Variable in verschiedenen Zielen in einer konjunktiven Anfrage erscheint, dann bezieht sie sich immer auf das selbe Objekt.

  ▷ `father(haran,X),male(X)`

- Eine konjunktive Anfrage ist die Folge eines Programms, wenn jedes Ziel eine Folge des Programms ist, wobei jede Variable mit dem selben Wert in verschiedenen Zielen ersetzt wird.

Regeln

- Regeln erlauben, neue Beziehungen mit Hilfe existierender Beziehungen zu definieren.
- Regeln sind Aussagen der Form:

$$A \leftarrow B_1, B_2, \ldots, B_n$$

  - ▷ A heißt Kopf der Regel.
  - ▷ Die Konjunktion von Zielen $B_1, B_2, \ldots, B_n$ heißt Rumpf der Regel.
  - ▷ Fakten sind Regeln im Speziallfall $n = 0$. Im Allgemeinen ist ein Programm eine endliche Menge von Regeln (statt eine Menge von Fakten).
- Fakten, Anfragen und Regeln heißen auch Horn Klauseln/Klauseln.

Regeln

- Variablen in Regeln sind (wie in Fakten) universell quantifiziert.
  - ▷ `son(X,Y)` $\leftarrow$ `father(Y,X), male(X).`
  - ▷ `daughter(X,Y)` $\leftarrow$ `father(Y,X), female(X).`
  - ▷ `grandfather(X,Y)` $\leftarrow$ `father(X,Z), father(Z,Y).`
- "$\leftarrow$" stellt logische Implikation dar
  $\Longrightarrow$ *Modus ponens* als Deduktionsregel:

Aus $R = (A \leftarrow B_1, B_2, \ldots, B_n)$ und $B'_1, B'_2, \ldots, B'_n$ folgt A'
wenn $A' \leftarrow B'_1, B'_2, \ldots, B'_n$ eine Instanz von $A \leftarrow B_1, B_2, \ldots, B_n$ ist.

- ▷ Identität und Instanziierung sind Spezialfälle des Modus ponens.

Prozedurale vs. logische Interpretation der Regeln

▷ `grandfather(X,Y)` ← `father(X,Z), father(Z,Y).`

- Prozedurale I.: *Um die Anfrage "ist X der GroßVater von Y?" zu beantworten, beantworte die konjunktive Anfrage "ist X der Vater von Z und Z der Vater von Y?".*
- Logische I.: *Für alle X, Y und Z, X ist der Großvater von Y, wenn X der Vater von Z und Z der Vater von Y ist.*

5.2 Logische Programme als deduktive Datenbanken

- Ausser Relationen enthält eine deduktive/logische Datenbank Regeln, die erlauben, neue Relationen aus bestehenden Relationen zu gewinnen.

  $\Longrightarrow$ Logische Programme können als deduktive Datenbanken betrachtet werden.

  ▷ Grundrelationen werden via Fakten spezifiziert.

  ▷ Regeln entsprechen der logischen Regeln.

Beispiele

- Als Basisrelationen nehmen wir `father/2`, `mother/2`, `male/1` und `female/1` an, wobei die Zahl die Stelligkeit der jeweiligen Relation angibt.
- Wir definieren neue Relationen mit Hilfe der bestehenden Relationen via Regeln:
  - `parent(Parent,Child)` $\leftarrow$
    `father(Parent,Child).`
    `parent(Parent,Child)` $\leftarrow$
    `mother(Parent,Child).`
    ($\longrightarrow$ Mehrere Regeln mit dem selben Kopf definieren eine Disjunktion.)
  - `procreated(Man,Woman)` $\leftarrow$
    `father(Man,Child), mother(Woman,Child).`

Beispiele

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),male(Brother).
```

- ▷ Problem: `brother(X,X)?` gilt für jedes männliche Kind `X`.
- ▷ Lösung: Wir nehmen an, es existiert ein vordefiniertes Prädikat `≠(Term1,Term2)`, das wahr ist, wenn `Term1` verschieden von `Term2` ist. Wir schreiben das Prädikat infixiert: `Term1 ≠ Term2`.

$\Longrightarrow$

```
brother(Brother,Sib) ←
   parent(Parent,Brother),parent(Parent,Sib),
   male(Brother),Brother≠Sib.
```

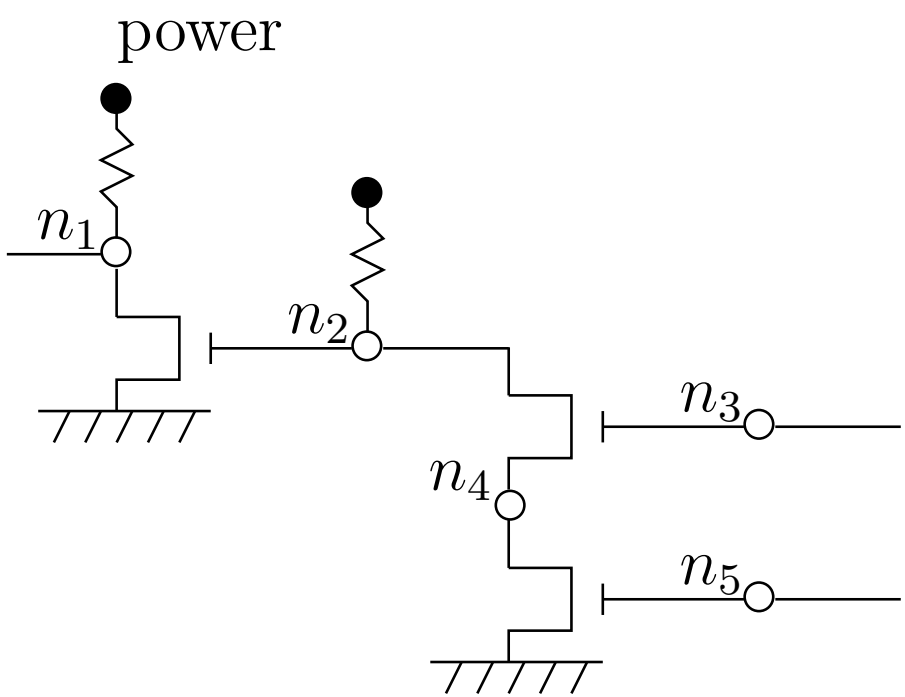
Beispiele

- `uncle(Uncle,Person)` $\leftarrow$
  `brother(Uncle,Parent),parent(Parent,Person)`
- `sibling(Sib1,Sib2)` $\leftarrow$
  `parent(Parent,Sib1),parent(Parent,Sib2),Sib1`$\neq$`Sib2.`
- `cousin(Cousin1,Cousin2)` $\leftarrow$
  `parent(Parent1,Cousin1),parent(Parent2,Cousin2),`
  `sibling(Parent1,Parent2).`

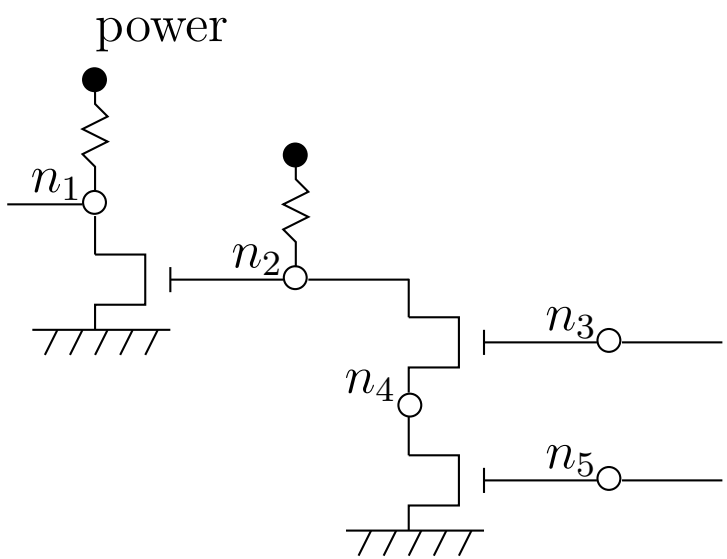
Beispielanfragen

- Sind Hanah und Isaac Geschwister: `sibling(hanah,isaac)?`
- Wer ist der Onkel von Hanah: `uncle(X,hanah)?`
- Welche Geschwisterpaare gibt es: `sibling(X,Y)?`

Beispiel



Beispiel

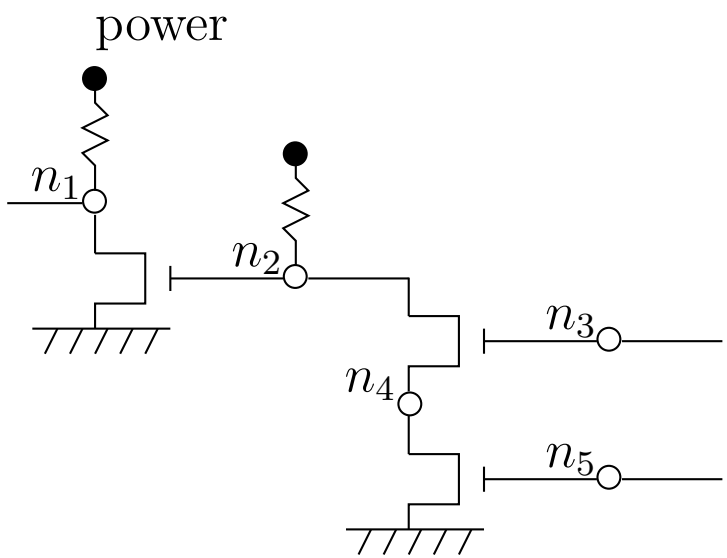



```
resistor(power,n1).
resistor(power,n2).
transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```

Beispiel

- ```
  not(Input,Output) ←
    transistor(Input,ground,Output),resistor(power,Output).
  ```
- ```
  nand(Input1,Input2,Output) ←
    transistor(Input1,X,Output),
    transistor(Input2,ground,X),
    resistor(power,Output).
  ```
- ```
  and(Input1,Input2,Output) ←
    nand(Input1,Input2,X),
    not(X,Output).
  ```
  ```

Beispielanfragen



`and(In1,In2,Out)?`

$\Longrightarrow$ `{In1=n3,In2=n5,Out=n1}`

Strukturierte Daten und Datenabstraktion

- Flache Darstellung einer Vorlesung:

```
course(compilerbau,montag,9,11,helmut,seidl,MW,1001).
```

  $\longrightarrow$ Kompakte aber unübersichtliche Darstellung

- Strukturierte D.:

```
course( compilerbau,
        zeit(montag,9,11),
        lecturer(helmut,seidl),
        room(MW,1001)).
```

  $\longrightarrow$ Abstraktion der für die jeweiligen Ziele unwesentlichen Details, z.B.:

```
lecturer(Lecturer,Course) ← course(Course,Time,Lecturer,Loc).
teaches(Lecturer,Day) ← course(Course,time(Day,S,F),Lecturer,Loc).
```

Rekursive Regeln

```
ancestor(Ancestor,Descendant) ←
      parent(Ancestor,Descendant)

ancestor(Ancestor,Descendant) ←
      parent(Ancestor,Person), ancestor(PersonDescendant)
```

5.3 Logische Programme zur Bearbeitung rekursiver Datenstrukturen

5.3.1 Bäume

- ```
  bintree(void)
  bintree(tree(Element,Left,Right)) ←
     bintree(Left), bintree(Right).
  ```

- ```
  member(X,tree(X,Left,Right)).
  member(X,tree(Y,Left,Right)) ← member(X,Left).
  member(X,tree(Y,Left,Right)) ← member(X,Right).
  ```

Bsp.: Baumisomorphie

- Zwei Bäume t_1 und t_2 sind isomorph, wenn t_2 durch eine Umordnung der Zweige in t_1 erhalten werden kann.
- Bäume in denen die Reihenfolge der Söhne unwichtig ist = ungeordnete Bäume

```
isotree(void,void).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Left2), isotree(Right1,Right2).

isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) ←
  isotree(Left1,Right2), isotree(Right1,Left2).
```

Bsp.: Substitution in Bäumen

- Problem: Ersetze im Baum T_1 alle Vorkommen von X durch Y.
- Lösung: Die Eingabedaten und das Ergebnisbaum T_2 sind Argumente eines Prädikats, `substitute(X,Y,`T_1`,`T_2`)`, definiert wie folgt:

```
substitute(X,Y,void,void).
substitute(X,Y,tree(Info,Left,Right),
                tree(Info1,Left1,Right1)) ←
  replace(X,Y,Info,Info1),
  substitute(X,Y,Left,Left1),
  substitute(X,Y,Right,Right1).

replace(X,Y,X,Y).
replace(X,Y,Z,Z) ← X ≠ Z.
```

5.3.2 Listen

- Listen sind ein Spezialfall von Binärbäumen und können syntaktisch definiert werden mit zwei Konstrukten:
  - ▷ Leere Liste: `[]`
  - ▷ Nicht-leere Liste: : `[X|Y]`

    mit `X` das erste Element und `Y` der Rest der Liste.
- Syntakische Konvention: `[a|[b|[c|[]]]]` ≡ `[a,b,c]`

Bsp.: member

```
member(X, [X|Xs]).
member(X, [Y|Ys]) ← member(X, Ys).
```

Bsp.: prefix, suffix, sublist

```
prefix([],Ys).
prefix([X|Xs],[X|Ys]) ← prefix(Xs,Ys).

suffix(Xs,Xs).
suffix(Xs,[Y|Ys]) ← suffix(Xs,Ys).

sublist(Xs,Ys) :- prefix(Ps,Ys), suffix(Xs,Ps).
```