Proseminar
"Grundlagen höherer Programmiersprachen"
Wintersemester 2002/03
(Kröger, Rauschmayer)

Logikprogrammierung

Verfasser:
Bettina Hikele
3. Semester Informatik
E-Mail: bettyx@arcor.de

- **Einführung in die Logikprogrammierung**
- **Deduktiver Informationsabruf**
- **Logik als Programm**

Einführung

Informatik	Mathematik
Imperatives Wissen	Deklaratives Wissen
("Wie geht das?")	("Was ist das?")

Bedeutung von Funktionen

Höhere Programmiersprachen:

→ Berechnung des Wertes von mathematischen Funktionen

→ Berechnung in einer Richtung

Ausdrucksorientierte Programmiersprachen (z. B. Lisp):

→ Ausdrücke "beinhalten mehr Information"

1. **Berechnung des Wertes von mathematischen Funktionen**
2. **Beschreibung eines Funktionswertes**

Vorteile einer funktionierenden Logikprogrammierung

→ **Sehr wirkungsvoll für die Programmierung**

→ **Verschiedene Lösungsmöglichkeiten für ein Problem**

→ **Funktionen verlaufen nicht nur in eine Richtung**

Nachteile der derzeitigen Logikprogrammierung

→ **Terminiert unter Umständen nicht oder anderes unerwünschtes Verhalten**

→ **Erfordert den Einsatz einer völlig neuen (parallelen) Rechnerarchitektur**

Interpretierer für eine Sprache der Logikprogrammierung:

- **Ein Teil zum Auswerten (für die Typklassifizierung)**
- **Ein Teil zum Anwenden (Implementation des Abstraktionsmechanismus, bzw. Regeln)**
- **Datenstruktur mit Bindungsrahmen**
- **Datenströme**

Als Sprache wird eine Anfragesprache verwendet.

→ Hilfreich beim Abrufen von Informationen aus den Datenbanken.

Deduktiver Informationsabruf

In der Logikprogrammierung werden Schnittstellen bereitgestellt, über die Informationen aus Datenbanken abgerufen werden können.

```
(address (Bit Ben) (Slumerville (Ridge Road) 10))
(job (Bit Ben) (computer wizard))
(salary (Bit Ben) 60000)
(supervisor (Bit Ben) (Warbucks Oliver))

(address (Hacker Alyssa) (Cambridge (Mass Ave) 78))
(job (Hacker Alyssa) (computer programmer))
(salary (Hacker Alyssa) 40000)
(supervisor (Hacker Alyssa) (Bit Ben))

(address (Fect Cy) (Cambridge (Ames Street) 3))
(job (Fect Cy) (computer programmer))
(salary (Fect Cy) 35000)
(supervisor (Fect Cy) (Bit Ben))

(address (Tweakit Lem) (Boston (Bay State Road) 22))
(job (Tweakit Lem) (computer technician))
(salary (Tweakit Lem) 25000)
(supervisor (Tweakit Lem) (Bit Ben))

(address (Reasoner Louis) (Slumerville (Pine Tree Road) 80))
(job (Reasoner Louis) (computer programmer trainee))
(salary (Reasoner Louis) 30000)
(supervisor (Reasoner Louis) (Hacker Alyssa))

(address (Warbucks Oliver) (Swellesley (Top Heap Road)))
(job (Warbucks Oliver) (administration big wheel))
(salary (Warbucks Oliver) 150000)

(address (Scrooge Eben) (Weston (Shady Lane) 10))
(job (Scrooge Eben) (accounting chief accountant))
(salary (Scrooge Eben) 75000)
(supervisor (Scrooge Eben) (Warbucks Oliver))

(address (Cratchet Robert) (Allston (N Harvard Street) 16))
(job (Cratchet Robert) (accounting scrivener))
(salary (Cratchet Robert) 18000)
(supervisor (Cratchet Robert) (Scrooge Eben))

(address (Aull DeWitt) (Slumerville (Onion Square) 5))
(job (Aull DeWitt) (administration secretary))
(salary (Aull DeWitt) 25000)
(supervisor (Aull DeWitt) (Warbucks Oliver))
```

Einfache Anfragen

- **Einträge werden nach einem bestimmten Muster in der Datenbank gesucht**
- **Ausgegeben werden alle dem Muster entsprechenden Einträge**

Darstellung der Mustervariablen:

```
?Variablenname
```

Darstellung eines Musters:

```
>(Symbol Mustervariable Liste)
```

Muster aus drei Elementen:

```
>(job ?x (computer programmer))
(job (Hacker Alyssa) (computer programmer))
(job (Fect Cy) (computer programmer))
```

Keine Mustervariable:

Stellt fest, ob das Muster in der Datenbank enthalten ist.

Mehrere Variablen:

```
>(adresse ?x ?y)
```

(Listet die Adressen aller Angestellten auf)

```
>(supervisor ?x ?x)
```

(Listet alle auf, die Vorgesetzter von sich selbst sind)

Elemente einer Liste:

```
>(job ?x (computer ?type))
(job (Bit Ben) (computer wizard))
```

Nicht ausgegeben wird:

```
(job  (Reasoner Louis)
      (computer programmer trainee))
```

(Die Liste enthält drei statt zwei Elemente)

Dotted-tail notation:

```
>((job ?x (computer . ?type))
(job  (Bit Ben) (computer wizard))
(job  (Reasoner Louis)
      (computer programmer trainee))
```

→ " . ?Variable" entspricht dem Rest einer Liste

Zusammengesetzte Anfragen

AND

```
>(and  (job ?person (computer programmer))
       (address ?person ?where))
(and (job (Hacker Alyssa)(computer programmer))
     (address (Hacker Alyssa)
              (Cambridge (Mass Ave) 78)))
(and  (job (Fect Cy) (computer programmer))
     (address (Fect Cy)
              (Cambridge (Ames Street) 3)))
```

Beide Einträge müssen mit dem Muster übereinstimmen!

OR

```
>(or  (supervisor ?x (Bit Ben))
      (supervisor ?x (Hacker Alyssa)))
(or (supervisor (Hacker Alyssa)(Bit Ben))
    (supervisor (Hacker Alyssa)
                (Hacker Alyssa)))
(or (supervisor (Fect Cy) (Bit Ben))
    (supervisor (Fect Cy) (Hacker Alyssa)))
(or (supervisor (Tweakit Lem) (Bit Ben))
    (supervisor (Tweakit Lem) (Hacker Alyssa)))
(or (supervisor (Reasoner Louis) (Bit Ben))
    (supervisor (Reasoner Louis)
                (Hacker Alyssa)))
```

Mindestens ein Eintrag muss mit

einem Teil des Musters übereinstimmen!

NOT

```
>(and (supervisor ?x (Bit Ben))
      (not (job ?x (computer programmer))))
```

(Findet alle Angestellten, deren Vorgesetzter nicht Ben Bit ist.)

Alle Zuweisungen, die nicht dem Muster entsprechen.

Regeln

Durch die Angabe von Regeln wird
eine Abstraktion der Anfrage erreicht.

Das Muster muss nicht explizit als Aussage
in der Datenbank enthalten sein.
Es kann auch eine implizite Aussage sein,
die durch eine Regel impliziert wird.

Beispiel:

```
>(rule (wheel ?person)
       (and  (supervisor ?middle-manager
              ?person)
             (supervisor ?x ?middle-manager)))
```

Diese Regel besagt, dass jeder ein "Hohes Tier" ist,
wenn er der Vorgesetzte eines Vorgesetzten ist.

Regeln können Teil anderer Regeln sein
und rekursiv definiert werden.

Beispiel:

```
>(rule (outranked-by ?staff-person ?boss)
       (or (supervisor ?staff-person ?boss)
           (and  (supervisor ?staff-person
                  ?middle-manager)
                 (outranked-by
                  ?middle-manager ?boss))))
```

Diese Regel besagt:
(ein Angestellter ist einem Chef unterstellt, wenn der Chef der Vorgesetzte des Angestellten ist)
oder (rekursiv)
(wenn der Vorgesetzte des Angestellten dem Chef unterstellt ist)

Logik als Programm

Wir definieren eine Operation "append":

- **Argumente: Zwei Listen**
- **Die Elemente der zwei Listen werden zu einer Liste zusammengefügt**
- **Definition erfolgt mit Hilfe des Listenkonstruktors "cons"**

```
(define  (append x y)
         (if (null ?x)
             y
             (cons (car x) (append (cdr x) y))
         )
)
```

Diese Prozedur beinhaltet folgende zwei Regeln:

1. Regel

(append x y) → y

Falls x eine leere Liste ist!

2. Regel

(append (cons u v) y) → (cons u z)

wobei

(append v y) → z

In der Logikprogrammierung wird die Prozedur "append" durch die Angabe dieser zwei Regeln programmiert.

(append-to x y z) **"append von x und y führt zu z"**

1. Regel

```
>(rule (append-to () ?y ?y))
```

2. Regel

```
>(rule  (append-to (?u . ?v) ?y (?u . ?z))
        (append-to ?v ?y ?z))
```

1. Beispiel

Berechnung von:

```
>(append l1 l2)
```

```
>(append-to (a b) (c d) ?y)
(append-to (a b) (c d) (a b c d))
```

2. Beispiel

Für welches y gilt:

```
>(append (a b) y)
(a b c d)
```

```
>(append-to (a b) ?y (a b c d))
(append-to (a b) (c d) (a b c d))
```

3. Beispiel

Alle l1 und l2 für die gilt:

```
>(append l1 l2)
(a b c d)

>(append-to ?x ?y (a b c d))
(append-to () (a b c d) (a b c d))
(append-to (a) (b c d) (a b c d))
(append-to (a b) (c d) (a b c d))
(append-to (a b c) (d) (a b c d))
(append-to (a b c d) () (a b c d))
```

4. Beispiel

```
(append-to (1 2) (3) ?z)
```

- **Unifikation**

```
(append-to (1 2) (3) (1 . z1))
```

- **Anfrage**

```
(append-to (2) (3) z1)
```

- **Unifikation**

```
(append-to (2) (3) (2 . z2)
```

- **Anfrage**

```
(append-to () (3) z2)
```

- **Unifikation**

```
(append-to () (3) (3))
```