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

Logikprogrammierung

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

Inhaltsverzeichnis

1. Einführung in die Logikprogrammierung

1.1. Informatik – Mathematik

1.1.1 Informatik

In der Informatik steht das imperative Wissen ("Wie geht das?") im Vordergrund. Es müssen Methoden angegeben werden, die den Lösungsweg eines Problems genau beschreiben. Höhere Programmiersprachen bieten dem Anwender bereits viel methodisches Wissen, das heißt es ist weniger Detailwissen über den Rechenvorgang nötig.

1.1.2. Mathematik

Im Gegensatz zum Informatiker, muss sich der Mathematiker mehr mit deklarativem Wissen ("Was ist das?") befassen. Dabei ist die Auswertungsreihenfolge nicht explizit festgelegt, und Berechnung von mathematischen Funktionen kann in mehrere Richtungen erfolgen.

1.2. Bedeutung von Funktionen

1.2.1. Höhere Programmiersprachen

In höheren Programmiersprachen erfolgt die Berechnung des Wertes einer mathematischen Funktion nur in einer Richtung, das heißt die Berechnung erfolgt mit wohldefinierten Ein- und Ausgaben. Das imperative Wissen dazu, muss vom Programmierer größtenteils selbst bereitgestellt werden.

1.2.2. Ausdrucksorientierte Programmiersprachen

Ausdrucksorientierte Programmiersprachen, wie zum Beispiel Lisp, können aus Ausdrücken mehr Information gewinnen. Die Ausdrücke sind nicht nur Mittel zur Berechnung eines Funktionswertes, sondern sie liefern auch eine Beschreibung des Funktionswertes.

1.3. Vorteile einer funktionierenden Logikprogrammierung

Eine funktionierende Logikprogrammierung ist sehr wirkungsvoll für die Programmierung. Sie bietet verschiedene Lösungsmöglichkeiten für ein Problem. Außerdem können Funktionen in mehreren Richtungen ausgewertet werden.
Es ist in der Logikprogrammierung zum Beispiel möglich, vom Ergebnis auf die möglichen Eingaben zu schließen.

1.4. Nachteile der derzeitigen Logikprogrammierung

Die derzeitige Logikprogrammierung verursacht noch einige Probleme. Programme terminieren unter Umständen nicht, oder führen zu anderem unerwünschten Verhalten. Außerdem wäre der Einsatz einer völlig neuen (parallelen) Rechnerarchitektur nötig, um eine möglichst effektive Arbeitsweise zu gewähren (Diese Notwendigkeit wird später an den Beispielen noch deutlicher werden).

1.5. Interpretierer für eine Sprache der Logikprogrammierung

Der Interpretierer setzt sich aus folgenden Komponenten zusammen:

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

Als Sprache wird eine Anfragesprache verwendet. Dies ist hilfreich beim Abrufen von Informationen aus den Datenbanken.

2. Deduktiver Informationsabruf

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

2.1. Beispieldatenbank

Ein "typischer" Eintrag in der Datenbank besteht aus dem Namen, einem Symbol und einer Liste.

Die folgende Beispieldatenbank dient uns als Grundlage für die Anfragen in den nachfolgenden Beispielen:

```
(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))
```

2.2. Einfache Anfragen

Die Einträge werden nach einem bestimmten Muster in der Datenbank gesucht.
Ausgegeben werden alle dem Muster entsprechenden Einträge.

2.2.1. Darstellung der Mustervariablen

Die Mustervariable wird "?" gefolgt vom Variablennamen geschrieben: `?Variablenname`

2.2.2. Darstellung eines Musters

Ein "typisches" Muster besteht aus einem Symbol, einer Mustervariablen und der Liste.

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

2.2.3. Muster aus drei Elementen

Das folgende Muster besteht aus drei Elementen, wobei eine Liste mit zwei Elementen enthalten ist:

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

2.2.4. Keine Mustervariable

Ein Muster kann auch ohne Mustervariable angegeben werden. Es wird dann festgestellt, ob das Muster genau so in der Datenbank enthalten ist.

2.2.5. Mehrere Variablen

Auch die Angabe von mehreren Variablen ist möglich.

2.2.5.1 Verschiedene Variablen

In einem Muster können mehrere verschiedene Variablen angegeben werden.

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

Diese Muster listet die Adressen aller Angestellten auf.

2.2.5.2 Gleiche Variablen

In einem Muster kann die selbe Variable mehrmals enthalten sein.
Für eine Übereinstimmung, muss der Inhalt in der Datenbank jedoch den selben Wert für gleiche Variablen aufweisen.

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

Listet alle Angestellten auf, die Vorgesetzter von sich selbst sind.

2.2.6. Elemente einer Liste

Das folgende Beispiel soll den Umgang mit Listen verdeutlichen:

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

In diesem Fall enthält die Liste genau zwei Elemente. - Die Mustervariable steht für ein Element.

Nicht ausgegeben wird deshalb:

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

Diese Liste enthält drei statt zwei Elemente und stimmt daher nicht mit dem Muster überein.

2.2.7. Dotted-tail notation

Falls man sich im Muster noch nicht auf die Anzahl der Elemente in der Liste festlegen will (oder kann), gibt es die Möglichkeit der "dotted-tail notation". Dadurch wird nur angegeben, dass es sich um eine Liste mit Rest handelt. Der Rest kann dabei beliebig groß sein.

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

Durch das Setzen eines Punktes gefolgt von der Variablen, wird der Rest der Liste angezeigt, allerdings ohne dabei die Länge festzusetzen.

2.3. Zusammengesetzte Anfragen

2.3.1. AND

Durch die Angabe von "and" wird festgelegt, dass der gesuchte Eintrag mit beiden Teilen des Musters übereinstimmen muss. Zuerst werden dabei alle Einträge in der Datenbank gesucht, die der ersten Anforderung entsprechen. Anschließend erfolgt die Ausgabe von allen Einträgen, die auch der zweiten Bedingung gerecht werden.

```
>(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)))
```

2.3.2. OR

Bei der Angabe von "or" wird jeder Eintrag als Ergebnis ausgeben, der mit mindestens einem der beiden Teile des Musters übereinstimmt.

```
>(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)))
```

Hier werden zuerst alle Personen in der Datenbank gefunden, die Ben Bit als Vorgesetzten haben. Anschließend wird die Datenbank nach allen Personen durchsucht, die Alyssa Hacker als Vorgesetzte haben.

2.3.3. NOT

Die Anfrage mit "not" findet alle Zuweisungen, die nicht dem Muster entsprechen.

```
>(not (supervisor ?x (Bit Ben)))
```

Diese Anfrage findet alle Angestellten, deren Vorgesetzter nicht Ben Bit ist.
Ausgegeben werden alle Angestellten der Firma, nachdem die Personen mit Bit Ben als Vorgesetzten ausgefiltert worden sind.

2.4. Regeln

Durch die Angabe von Regeln wird eine Abstraktion der Anfrage erreicht. Das Muster muss dann nicht explizit als Aussage in der Datenbank enthalten sein. Es kann auch eine implizite Aussage sein, die durch eine Regel impliziert wird.

2.4.1. Beispiel 1

```
>(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.

2.4.2. Beispiel 2

Regeln können Teil anderer Regeln sein und rekursiv definiert werden.

```
>(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

3. Logik als Programm

Wir definieren zuerst funktional eine Operation "append":

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

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

Diese Prozedur beinhaltet folgende zwei Regeln:

1. (append x y) → y / Falls x eine leere Liste ist!
2. (append (cons u v) y) → (cons u z) / falls (append v y) → z

Nun wollen wir die selbe Prozedur logisch (als "append-to") definieren. In der Logikprogrammierung wird "append" durch die Angabe der zwei Regeln programmiert:

`(append-to x y z)` ("append-to von x und y führt zu z" bzw. (append-to l1 l2 l1+l2))

1. Regel:

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

2. Regel:

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

Die erste Regel besagt, dass nur die zweite Liste y ausgegeben wird, falls die erste Liste leer ist. Die zweite Regel spaltet das erste Element (?u) der Liste x ab und gibt an, dass es das erste Element der Ausgabeliste ist (mit dem Rest ?z). Anschließend wird der Rest der Liste x mit (append-to ?v ?y ?z) abgearbeitet, bis die Liste leer ist, und anschließend die erste Regel aufgerufen.

In den folgenden Beispielen werden die neuen Anfragemöglichkeiten aufgezeigt. Durch die Gegenüberstellung von "append" und "append-to", sollen die Vorteile der Logikprogrammierung verdeutlicht werden.

3.1. Beispiel 1

Berechnung von:

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

In diesem Beispiel werden zwei Listen (a b) und (c d) zu (a b c d) aneinander gehängt.

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

3.2. Beispiel 2

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

Durch die Angabe der Regeln können wir nun erfragen, welche Liste y benötigt wird, um zusammen mit (a b) die Liste (a b c d) zu erhalten.

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

3.3. Beispiel 3

Wir können mit der Funktion "append-to" sogar alle möglichen Listenpaare erfragen, die als Ergebnis die Liste (a b c d) liefern.

```
>(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))
```

3.4. Beispiel 4

Im folgenden Beispiel wird dargestellt, wie die Anfrage im Einzelnen ausgewertet wird. Bevor eine Anfrage erfolgt, wird zuerst eine Verallgemeinerung des Musters (Unifikation) vorgenommen. Anschließend wird nach der dazu passenden Regel gesucht. Dieser Vorgang wird solange wiederholt, bis die Auswertung beendet ist.

```
(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))
```