Prof. Dr. Christoph Bockisch
Steffen Dick
Fachbereich Mathematik und Informatik
AG Programmiersprachen und -werkzeuge

Philipps Universität Marburg

Präsenzübungen zur Vorlesung
Deklarative Programmierung: Sommersemester 2018
Nr. 10

Aufgabe 10.1: Listiges Prolog

Schreiben Sie Listenprädikate:

a) `contains(X,L)`, welches erfüllt ist, falls `X` in `L` enthalten ist.

b) `append(L1,L2,L3)`, welches erfüllt ist, falls `L3` durch anhängen von `L2` an `L1` entsteht.

c) Welches Ergebnis produzieren die Anfragen "`contains(2,L)`", "`append(L1,[2,3],[1,2,3,4])`" und "`append(L1,L2,[1,2,3,4])`"?

Aufgabe 10.2: Prolog Thing

Betrachten Sie die folgende Implementierung von binären Bäumen in Prolog:

```
btree(void).
btree(tree(Element,Left,Right)) :- btree(Left),btree(Right).
```

Schreiben Sie die folgenden Prädikate:

a) `is_leaf(Element,Tree)`, das erfüllt ist, falls `Element` in einem Blattknoten von Tree gespeichert ist. Ein Knoten ist ein Blattknoten, falls beide Nachfolger `void` sind.

b) `subtree(S,T)`, das erfüllt ist, falls `S` ein Teilbaum von `T` ist.

Aufgabe 10.3: Nickel-Metallhydrid

Die Fakultät ($n!$) is das Produkt aller natürlicher Zahlen bis n. Nachfolgend ist eine (strukturell) rekursive Definition der Fakultät gegeben.

$$n! = \begin{cases} 1, & n = 0 \\ n \cdot (n-1)!, & n = 1 \end{cases}$$

Wenn man die Zahlen in absteigender Reihenfolge multipliziert, sieht man direkt, dass sich die Fakultät auch mittels eines Akkumulators berechnen lässt:

$$n! = \prod_{i=1}^{n} i = \left(\left(\left(\left(n \cdot (n-1)\right) \cdot (n-2)\right) \cdot ...\right) \cdot 1\right)$$

a) Implementieren Sie die Relation `fak1(N,R)`, die erfüllt ist, wenn R die Fakultät von N ist. Verwenden Sie hierzu strukturelle Rekursion.

b) Implementieren Sie die Relation `fak2(N,R)`, die unter denselben Umständen erfüllt ist. Jedoch verwenden Sie für deren Implementierung eine Hilfsfunktion `fak2_akk(N,Akk,R)`. Mit welchem Wert für `Akk` muss `fak2(N,R)` diese Hilfsfunktion initial aufrufen?

Aufgabe 10.4: Enter the Matrix

Machen Sie sich zunächst mit den Begriffen Matrix (`https://de.wikipedia.org/wiki/Matrix_(Mathematik)`) und Vektor (`https://de.wikipedia.org/wiki/Vektor`) vertraut.
Lösen Sie diese Aufgabe mit Racket und mit Prolog.

a) Definieren Sie geeignete Datenstrukturen für Matrizen und Vektoren. Nutzen Sie dafür in Racket `define-type`.

b) Definieren Sie nun eine Funktion `(multiply x y)` bzw. eine Prozedur `multiply(X,Y,Z)`, die die beiden Argumente x und y miteinander multipliziert (Racket) bzw. die erfüllt ist, wenn Z das Ergebnis der Multiplikation von X und Y ist (Prolog). Gehen Sie hierbei davon aus, dass es sich bei "x" ("X") um eine Matrix oder einen Vektor handelt und "y" ("Y") ein Skalar ist.

c) Erweitern Sie `multiply` dahingehend, dass zusätzlich zur skalaren Multiplikation auch Vektoren und Matrizen miteinander multipliziert werden können. Ermitteln Sie in Racket hierzu mit `type-case` die korrekte Berechnungsformel und wenden Sie diese an.

Hinweis: Fügen sie "(require 2htdp/abstraction)" zu Ihrem Programm hinzu.

Aufgabe 10.5: It ain't Racket-science!

Eine komplexe Zahl ist ein Paar (a, b) von reellen Zahlen. a nennt man den Realteil und b den Imaginärteil. Man kann dann wie folgt rechnen:

$$(a, b) + (c, d) = (a + c, b + d)$$
$$(a, b) * (c, d) = (a * c - b * d, a * d + b * c).$$

a) (Bearbeiten Sie diese Aufgabe mit Racket und mit Prolog.) Definieren Sie eine geeignete Datenstruktur zur Repräsentation komplexer Zahlen und implementieren Sie die Funktionen `(complex/add c1 c2)` (bzw. die Prozedur `complex/add(C1, C2, C3)`) und `(complex/mult c1 c2)` (bzw. `complex/mult(C1, C2, C3)`) zur Addition und Multiplikation.

b) (Bearbeiten Sie diese Aufgabe nur in Racket.) Implementieren Sie eine Funktion `(complex->string c)`, die eine komplexe Zahl als String repräsentiert. Die Regeln dafür sind wie folgt:

$$(\text{complex} \rightarrow \text{string } (a, b)) = \begin{cases} 0, \text{ falls } a = b = 0 \\ a, \text{ falls } a \neq 0 \wedge b = 0 \\ bi, \text{ falls } a = 0 \wedge b \neq 0 \\ a + bi, \text{ falls } a \neq 0 \wedge b \neq 0 \end{cases}$$

Nutzen Sie das `match`-Konstrukt um diese Fallunterscheidung umzusetzen!

c) (Bearbeiten Sie diese Aufgabe mit Racket und mit Prolog.) Komplexe Zahlen können neben dieser Darstellung auch in der sog. Polarform dargestellt werden (vgl. `https://de.wikipedia.org/wiki/Komplexe_Zahl`). Definieren Sie einen weiteren Typ: `(polar r phi)`, der komplexe Zahlen in Polarform repräsentiert. Implementieren Sie zudem Funktionen zur Konvertierung zwischen den beiden Darstellungen und führen Sie die Addition, Multiplikation und String-repräsentation auf Ihre Implementierung aus den Aufgabenteilen a) und b) zurück. Nutzen Sie pattern-matching zur Fallunterscheidung in den Konvertierungsfunktionen (falls angebracht).