Klausur „Programmierparadigmen (PGP)“ (Sommer 2009)

31. Juli 2009, 9:30 – 11:30

Aufgabe 1 [Gesamtpunktzahl: 12]

Ein ungerichteter, gewichteter Graph sei in **Prolog** über eine Liste mit dreielementigen Listen für seine Kanten und ihre Gewichte dargestellt.

Beispiel:

```
[[a,b,12],[a,c,34],[a,e,78],[b,d,55],[b,e,32],[c,d,61],[c,e,44],[d,e,93]]
```

Um für Graphen in dieser Darstellung zum Beispiel den Algorithmus von Prim implementieren zu können, sind einige **Prolog-Prädikate** nützlich, die im folgenden zu entwickeln sind.

a) Entwickeln Sie ein Prolog-Prädikat `edgeIn(Graph,[N1,N2],Edge)`, das genau dann zutrifft, wenn die beiden Knoten `N1` und `N2` im ungerichteten Graphen `Graph` mit der Kante `Edge` verbunden sind. Dabei soll `Edge` mit `[N1,N2,C]` unifiziert werden.

Beispiele: für `Graph` unifiziert mit der Listendarstellung von oben soll sich ergeben:

```
?- edgeIn(Graph,[e,d],Edge).
Edge = [e, d, 93]

?- edgeIn(Graph,[a,c],Edge).
Edge = [a, c, 34]

?- edgeIn(Graph,[e,g],Edge).
No
```

b) Entwickeln Sie ein Prolog-Prädikat `fromGraph(Graph,Pairs,FromGraph)`, das genau dann zutrifft, wenn `FromGraph` die Liste derjenigen Kanten aus dem `Graph` ist, die zu Paaren von Knoten aus der Liste `Pairs` existieren.

Beispiel: für `Graph` unifiziert mit der Listendarstellung von oben soll sich ergeben (Reihenfolge der Kanten kann abweichen):

```
?- fromGraph(Graph,[[a,b],[a,d],[a,e],[c b],[c,d],[c,e]],FromGraph).

FromGraph = [[c, e, 44], [c, d, 61], [a, e, 78], [a, b, 12]]
```

c) Im folgenden soll aus einer unsortierten Liste von Kanten diejenige mit minimalem Gewicht bestimmt werden.

Sie können dazu verwenden, dass in Prolog das Meta-Prädikat `predsort(Pred,List,Sorted)` vordefiniert ist, das zutrifft, wenn die Liste `Sorted` die gemäss dem Vergleichsprädikat `Pred` sortierte Version von `List` ist.

Vergleichsprädikate für `predsort` sind organisiert wie das vordefinierte `compare(Delta,E1,E2)`, bei dem `Delta` unifiziert wird mit einer der Konstanten <, > oder =, je nachdem, in welchem Verhältnis `E1` zu `E2` steht.

Beispiel:

```
?- compare(Delta,3,4).
Delta = <
```

c1) Wie lässt sich ein analoges Vergleichsprädikat `edgeCompare(Delta,E1,E2)` für Kanten eines Graphen in der obigen Darstellung definieren?

c2) Wie lässt sich dann durch Anwendung von `predsort` das Prädikat `minimumEdge(Edges,Min)` definieren, das genau dann zutrifft, wenn `Min` diejenige der Kanten aus `Edges` mit dem kleinsten Gewicht ist? Beispiel:

```
?- minimumEdge([[c, e, 44], [c, a, 34], [d, e, 93], [d, b, 55]],Min).
Min = [c, a, 34]
```

Aufgabe 2 [Gesamtpunktzahl: 12]

Ein ungerichteter, gewichteter Graph sei in **Scheme** über eine Liste mit dreielementigen Listen für seine Kanten und ihre Gewichte dargestellt.

Beispiel:

```
(define graph1
 '((a b 12)(a c 34)(a e 78)(b d 55)(b e 32)(c d 61)(c e 44)(d e 93)))
```

Um für Graphen in dieser Darstellung zum Beispiel den Algorithmus von Prim implementieren zu können, sind einige **Scheme-Funktionen** nützlich, die im folgenden – zusammen mit ggf. erforderlichen, aber nicht bereits vordefinierten Hilfsfunktionen – zu entwickeln sind.

a) Definieren Sie eine **Scheme-Funktion** `nodes`, die zu einem in obiger Darstellung gegebenen Graphen die Liste der Knoten zurückliefert.

Beispiel (Reihenfolge der Knoten kann abweichen):

```
> (nodes graph1)
(a b c e d)
```

b) Definieren Sie ein **Scheme-Prädikat** `edgeIn`, das für einen Graphen und zwei Knoten genau dann zutrifft, wenn es im Graphen eine Kante zwischen den beiden Knoten gibt.

c) Definieren Sie eine **Scheme-Funktion** `(candEdges graph tl rl)`, welche die Liste derjenigen Kanten des Graphen `graph` bestimmt und zurückliefert, bei denen der Startknoten aus der Knotenmenge `tl` und der Zielknoten aus der Knotenmenge `rl` kommt.

Beispiele (Reihenfolge in Ergebnislisten kann abweichen):

```
> (candEdges graph1 '(a b) '(c d e))
((a e 78) (a c 34) (b e 32) (b d 55))
> (candEdges graph1 '(a b c) '(d e))
((a e 78) (b e 32) (b d 55) (c e 44) (c d 61))
```

Aufgabe 3 [Gesamtpunktzahl: 8]

Aus der Vorlesung kennen Sie die Definition der **Scheme-Funktion** `curry`. Mit dieser *Funktion höherer Ordnung* lassen sich auch in Scheme partielle Anwendungen realisieren.

```
(define curry
  (lambda (f) (lambda (a) (lambda (b) (f a b)))))

> (map ((curry +) 3) '(1 2 3))
(4 5 6)
```

a) Definieren Sie in **Scheme** die Funktion `(filter pred liste)`. Diese *Funktion höherer Ordnung* bestimmt zu einer Liste `liste` die Liste derjenigen Elemente, auf die das Prädikat `pred` zutrifft. (Bemerkung: Dabei bleiben im Ergebnis enthaltene Elemente in ihrer ursprünglichen Reihenfolge).

Beispiel:

```
> (filter odd? '(1 2 3 4 5))
(1 3 5)
> (filter even? '(1 2 3 4 5))
(2 4)
```

b) Definieren Sie in **Scheme** die *Funktion höherer Ordnung* `(flip foo)`, die der zweiargumentigen Funktion `foo` diejenige Funktion zuordnet, die sich wie `foo` verhält, aber ihre Argumente in vertauschter Reihenfolge nimmt.

Beispiel:

```
> ((flip cons) '(a b c) 'vorne)
(vorne a b c)
```

c) Gegeben sei eine Liste `ll` mit Listen als Elementen. Durch eine Anwendung von `map` soll an jede der Elementlisten dieselbe Ergänzung `erg` angehängt werden.

Beispiel:

```
> (define ll '((a b) (c d) (e f)))

> (define erg '(y z))
```

Anhängen von `erg` an die Elemente von `ll` liefert

```
> (map <...> ll)
((a b y z) (c d y z) (e f y z))
```

c1) Ersetzen Sie <...> durch eine geeignete anonyme Funktion.

c2) Ersetzen Sie <...> durch eine Kombination von u.a. `flip` und `curry`.

Aufgabe 4 [Gesamtpunktzahl: 8]

a) Gegeben seien die folgenden Definitionen in **Scheme**.

```
(define x (list 'a 'b))

(define z1 (cons x x))

(define z2 (cons (list 'a 'b)(list 'a 'b)))

(define (set-to-wow! x)
        (set-car! (car x) 'wow)
        x)
```

Damit ergibt sich dann die folgende Interaktion:

```
> z1
((a b) a b)
> z2
((a b) a b)
> (set-to-wow! z1)
((wow b) wow b)
> (set-to-wow! z2)
((wow b) a b)
> z1
((wow b) wow b)
> z2
((wow b) a b)
```

Erläutern Sie anhand von Box-und-Pointer-Darstellungen, wie es zu der unterschiedlichen Wirkung von `set-to-wow!` bei `z1` und `z2` kommt.

b) Definieren Sie eine **Scheme-Funktion** `(remdups! liste)`, bei der *destruktiv* aus `liste` alle *unmittelbar aufeinanderfolgenden Mehrfachvorkommen* von Elementen entfernt werden.

Beispiel:

```
> (define l1 '(1 2 2 2 3 3 1 2))
> l1
(1 2 2 2 3 3 1 2)
> (remdups! l1)
(2)
> l1
(1 2 3 1 2)
```