



Technische Universität Berlin

Software and Embedded Systems Engineering Group

Prof. Dr. Sabine Glesner

www.sese.tu-berlin.de

Sekr. TEL 12-4

Ernst-Reuter-Platz 7

10587 Berlin











Softwaretechnik und Programmierparadigmen WiSe 2023/2024

Prof. Dr. Sabine Glesner
Simon Schwan
Julian Klein

Übungsblatt 8

Aufgabe 1: Listen und Listenfunktionen

- a) Schreibt ein Prädikat `printList(L)`, mit dem eine Liste ausgegeben werden kann. 
- b) Schreibt ein Prädikat, mit dem eine Liste invertiert werden kann. 
- c) Das Prädikat `middle(L,Erg)` soll das mittlere Element einer Liste mit einer ungeraden Anzahl Elemente finden. Löst möglichst effizient. Was passiert, wenn man das Prädikat auf eine Liste mit gerader Anzahl Elemente anwendet? 
- d) Mit dem Prädikat `slice(LIn,IL,IR,LOut)` soll eine Sub-Liste von `LIn` mit den Elementen von Index `IL` bis `IR` (einschließlich, Indizes starten bei 0) in `LOut` erhalten werden. 
- e) Das Prädikat `route(X,Y,V)` soll nun die Verbindungen zwischen Städten in unserer Faktenbasis in der richtigen Reihenfolge ausgeben. Dafür müssen wir uns die besuchten Städte in der Liste `V` „merken“. Es soll außerdem sichergestellt werden, dass keine Stadt mehrfach besucht wird. Somit sollen keine unendlichen Routen mehr möglich sein.  
- f) `allRoutes(X,Y,ResultList)` soll alle möglichen Routen zwischen zwei Städten und ihre Längen in `ResultList` aufsammeln. Dazu muss die Funktion `Route` so umgeschrieben werden, dass deren Ergebnisse nicht ausgegeben werden, sondern in Variablen zur Verfügung stehen. Es können vordefinierte Prädikate verwendet werden. 
- g) Ziel der Übung ist es natürlich, die kürzeste Route zu finden. Erstellt das Prädikat `shortestRoute(X,Y,Shortest)`, mit dem die kürzeste Route und ihre Länge ermittelt werden kann. 

Lösung:

```
1  /* FAKTEN */
2
3  street(hamburg,berlin,300).
4  street(berlin,muenchen,700).
5  street(muenchen,frankfurt,300).
6  street(frankfurt,berlin,500).
7  street(muenchen,stuttgart, 300).
8  street(stuttgart,koeln, 250).
9  street(frankfurt,koeln, 200).
10
11 /* REGELN */
12 directConnection(X,Y,S) :- street(X,Y,S) ; street(Y,X,S).
13
14 printList([ ]) :- true.
15 printList([H|T]) :- write(H),write(" "),printList(T).
16
17 rev([ ],Rev,Result) :- Result=Rev.
18 rev([H|T],Rev,Result) :- rev(T,[H|Rev],Result).
19 /* wrapper fuer rev */
20 revertList(L,R) :- rev(L,[ ],R).
21
22 middle(L,Erg) :- middleH(L,L,Erg).
23 middleH([H|_], [_], H).
24 middleH([_|LR], [_,_|LRFast], V) :- middleH(LR,LRFast,V).
25
26 slice([ ],_,_,[ ]).
27 slice([H|_],0,0,[H]).
28 slice([LInH|LInT],0,IR,[LInH|Rest]) :- IR > 0, IR1 is IR -1,
29                                     slice(LInT,0,IR1, Rest), !.
30 slice([_|LInT],IL,IR,Rest) :- IL > 0, IL1 is IL -1, IR1 is IR -1,
31                               slice(LInT,IL1,IR1, Rest).
32
33 /* Route mit Zyklenpruefung: hier wird die Route am Ende gedruckt
34    (richtige Reihenfolge) */
35 route(X,Y,Visited) :- \+ member(X,Visited), \+ member(Y,Visited),
36                      directConnection(X,Y,_),
37                      revertList([Y,X|Visited],FinalRoute)
38                      printList(FinalRoute).
39 route(X,Z,Visited) :- \+ member(X,Visited), directConnection(X,Y,_),
40                      route(Y,Z,[X|Visited])).
41
```

```

42  /* Alternative (fuer allRoutes und shortestRoute): FinalRoute
43     erhaelt gefundene Route (richtige Reihenfolge) und FinalDist
44     die Laenge. So koennen diese Werte weiterverwendet werden. */
45  route2(X,Y,Visited,Dist,[X,Y]) :- \+ member(X,Visited),
46                                     \+ member(Y,Visited),
47                                     directConnection(X,Y,Dist).
48  route2(X,Z,Visited,Dist,[X|Route1]) :-
49      \+ member(X,Visited),
50      directConnection(X,Y,D),
51      route2(Y,Z,[X|Visited], Dist1, Route1), Dist is Dist1 + D.
52
53  allRoutes(X,Y,ResultList) :-
54      findall((Route,Distance), route2(X,Y,[ ],Distance,Route),ResultList).
55
56  shortestRoute(X,Y,Shortest) :-
57      setof((Distance,Route), route2(X,Y,[ ],Distance,Route),[Shortest|_]).
58
59  /* QUERIES
60
61  ?- printList([hallo,1,[1,2,3,4,5],komplex(eins,zwei)]).
62  hallo 1 [1,2,3,4,5] komplex(eins,zwei)
63  true.
64
65  ?- revertList([1,2,3,4,5], L).
66  L = [5, 4, 3, 2, 1].
67
68  ?- route(berlin, muenchen, [ ]).
69  berlin muenchen
70  true ;
71  berlin frankfurt muenchen
72  true ;
73  berlin frankfurt koeln stuttgart muenchen
74  true ;
75  false.
76
77  ?- route2(berlin,muenchen,[ ],D,R).
78  R = [berlin, muenchen],
79  D = 700 ;
80  R = [berlin, frankfurt, muenchen],
81  D = 800 ;
82  R = [berlin, frankfurt, koeln, stuttgart, muenchen],
83  D = 1250 ;
84  false.

```

```

85
86  ?- allRoutes(berlin,muenchen,L).
87  L = [ ([berlin, muenchen], 700), ([berlin, frankfurt, muenchen], 800),
88        ([berlin, frankfurt, koeln, stuttgart, muenchen], 1250)].
89
90  ?- shortestRoute(berlin,muenchen,L).
91  L = (700, [berlin, muenchen]).
92  */





```

Aufgabe 2: 4 Wikinger (Zusatzaufgabe)

Vier Wikinger sollen im Dunkeln in höchstens einer Stunde über eine Brücke gehen, haben dabei aber ein Problem: Die Brücke kann maximal nur zwei von ihnen tragen, außerdem haben sie nur eine Fackel dabei und es kann niemand ohne Fackel über die Brücke. Die Wikinger sind außerdem unterschiedlich schnell, sie brauchen zum Überqueren die folgenden Zeiten: (5min, 10min, 20min, 25min). Löst das Rätsel mithilfe von Prolog.

Aufgabe 3: Metriken

Ihr habt den Code einer Funktion in JAVA erhalten und sollt dessen Qualität überprüfen. Die Funktion `createOrder(int customerID, int productID, Instant created)` soll im System einen Auftrag erstellen. Als erstes wird geprüft, ob die Kunden- und Kundinnennummer (`customerID`) und Warennummer (`productID`) im System vorhanden sind, und ob das Erstelldatum (`created`) gültig ist. Die Kunden und Kundinnen können außerdem für bestimmte Waren Rabatte sammeln, was vom System automatisch überprüft wird. Wurden alle Daten korrekt eingegeben, wird ein Auftrag im System erstellt.

- Bestimmt die LOC- und NCLOC-Werte der Implementierung. Was sagen sie über die Komplexität aus? 
- Erstellt zu der angegebenen Implementierung den Kontrollflussgraph. Das Werfen einer Exception kann dafür wie ein return-Statement behandelt werden. 
- Zählt alle möglichen Ausführungspfade des Programmes. Was sagt diese Zahl über die Komplexität aus? 
- Berechnet die Zyklomatische Komplexität nach McCabe. 

```

1 public Order createOrder(int customerID, int productID,
2     Instant created) throws ShopInputException {
3
4     // handle invalid inputs
5     if(!customers.containsKey(customerID)){
6         throw new ShopInputException(
7             ShopInputException.INVALID_CUST_NR);
8     }
9     if(!products.containsKey(productID)){
10        throw new ShopInputException(
11            ShopInputException.INVALID_PROD_NR);
12    }
13    if(created == null){
14        throw new ShopInputException(
15            ShopInputException.INVALID_DATE);
16    }
17
18    // get customer and product for given IDs
19    Customer c = customers.get(customerID);
20    Product p = products.get(productID);
21    // create new order
22    Order order = new Order(customerID, productID,
23        p.getPrice(), created);
24
25    if((c.isPriorityCustomer() & p.isDiscountable())
26        || isAnniversary(created)) {
27        order.setDiscount(true);
28        while(!c.getDiscount().isEmpty()) {
29            order.discount(c.getDiscount().removeFirst());
30        }
31    }
32    return order;
33 }

```

Lösung:

- a) Nach Definition aus der VL sind es 33 für LOC, und 27 für NCLOC. Für sich allein sagt dies nicht viel über die Komplexität. Interessanter ist es um z.B. erstmal grob Funktionen im Code nach ihrem Umfang zu sortieren, oder Projekte über ihren Verlauf oder den Umfang ihrer Komponenten zu bewerten.
- b) Siehe Abbildung 1. Zum Erstellen kann auch das Eclipse-Tool verwendet werden, das geht allerdings unter Umständen etwas anders mit den Statements um.
- c) Das Programm hat eine nichttriviale Schleife – und wie wir oben gelernt haben, wissen wir daher also nicht, wie viele Pfade es gibt (Vielleicht unendlich, wenn sie nicht terminiert). Das ist natürlich schade, denn diese Zahl wäre wirklich sehr interessant für verschiedene weitere Berechnungen wie z.B. die Worst-Case Execution Time (WCET).
- d) McCabe: $\#edges - \#nodes + 2 = 21 - 16 + 2 = \mathbf{7}$ ($\#conditions + 1 = 6 + 1 = 7$)

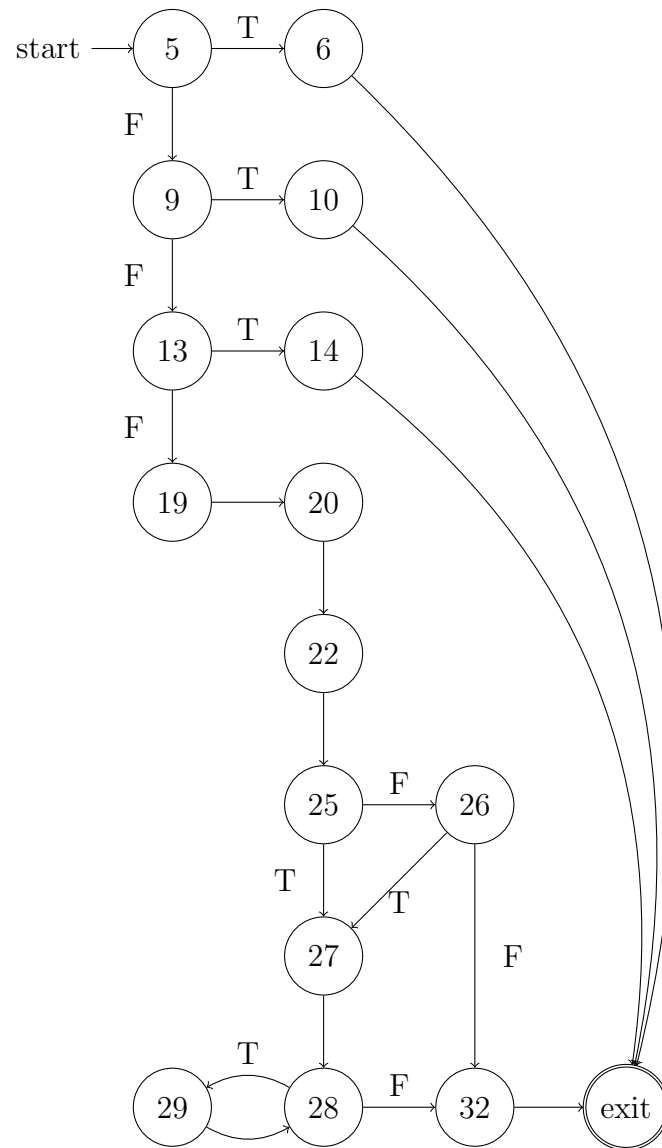


Abbildung 1: Kontrollflussgraph für `createOrder`.