Institut für Wissenschaftliches Rechnen
Technische Universität Braunschweig
Prof. Hermann G. Matthies, Ph. D.
Dr. Elmar Zander

Sommersemester 2010
14. November 2014

Weiterführendes Programmieren
Zeichenketten-Interpretation und Graphenalgorithmen
Aufgabenblatt 4

Interpretation von Sprachen

Bei der Entwicklung von Berechnungsprogrammen, und interaktiver Software ist es häufig notwendig komplexe Benutzereingaben zu interpretieren und in eine Systemantwort des Softwaresystems zu überführen. So stellen Eingaben auf der Ihnen bereits bekannten Kommandozeile von UNIX eine Interaktion zwischen Benutzer und einem Sprachinterpreter dar.
Beispiele wie das Einlesen komplexer Datensätze aus Dateien oder die Entwicklung von Anwendungssoftware für das Internet erfordern den Umgang mit Zeichenketten, deren Manipulation oder Interpretation.
In dieser Aufgabe soll der Umgang mit Zeichenketten und insbesondere deren Interpretation durchgenommen werden. Für die Interpretation einfacher Ausdrücke kann man sehr leicht einfache Interpreter schreiben. Erst bei komplizierteren Ausdrücken, wie sie z.B. bei der Interpretation einer Programmiersprache aufteten, sollte man für die Implementierung eines Interpreters unbedingt Werkzeuge wie `lex` oder `yacc` verwenden, welche z.B im Buch *lex & yacc – Die Profitools zur lexikalischen und syntaktischen Textanalyse* von Helmut Herold beschreiben werden.
In dieser Aufgabe wollen wir nicht so weit gehen und auf eine sehr einfache Sprache interpretieren. Bei der Beschreibung solcher formalen Sprachen ist es üblich die sogenannte Backus-Naur-Form (BNF) (`http://de.wikipedia.org/wiki/Backus-Naur-Form`) zu verwenden.

Sprache zur Beschreibung von Formalen Sprachen

In der Metasprache BNF (Metasprache bedeutet soviel wie “Sprache zur Beschreibung von Sprache”) kann man sogenannte Produktionen einsetzen, welche definieren, welche Form ein bestimmter Ausdruck annehmen kann. Beispiel:

```
<Ziffer ausser Null> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Eine *Ziffer außer der Null* ist eine 1, oder eine 2, oder eine 3,... Diese sogenannte *Produktionsregel* beschreibt, was eine Ziffer außer der Null ist. Es lassen sich auch Sequenzen definieren, bei denen bestimmte Symbole in einer bestimmten Reihenfolge auftreten:

```
<Ziffer>              ::= 0 | <Ziffer ausser Null>
<Zweistellige Zahl>   ::= <Ziffer ausser Null> <Ziffer>
<Zehn bis Neunzehn>   ::= 1 <Ziffer>
<Zweiundvierzig>      ::= 42
```

Eine Ziffer ist also eine 0 oder eine “Ziffer außer Null”. Eine zweistellige Zahl ist eine “Ziffer außer Null” gefolgt von einer Ziffer. Zweiundvierzig ist eine 4 gefolgt von einer 2.
Wiederholungen müssen in BNF über Rekursionen definiert werden. Eine Ableitungsregel kann dazu auf der rechten Seite das Symbol der linken Seite enthalten, etwa:

```
<Ziffernfolge> ::= <Ziffer> | <Ziffer> <Ziffernfolge>
```

Lies: Eine Ziffernfolge ist eine “Ziffer” oder eine “Ziffer gefolgt von einer Ziffernfolge”.
Mit Hilfe dieser Schreibweise kann man auf sehr einfache Weise die Struktur von Ausdrücken beschreiben. Wir wollen die BNF auf diesem Aufgabenblatt dazu benutzen eine sehr einfache Sprache zur Beschreibung von Graphen zu definieren. Doch nun wollen wir zunächst Graphen vorstellen, die wir später mit einer Sprache beschreiben wollen.

Graphen zur Modellierung von Wegen

Ein Graph im Sinne der mathematischen Graphentheorie besteht aus Elementen einer Menge (die sogenannten Knoten des Graphs), zwischen denen direkte Verbindungen bestehen können (die sogenannten Kanten des Graphs). Mathematisch ausgedrückt ist ein Graph G ein Tupel (V, E), wobei V eine Menge von Knoten und E eine Menge von Kanten bezeichnet. Dieser Definition genügt folgendes einfache Beispiel:

$$G_1 = (\{1, 2, 3\},\ \{(1, 2), (3, 1), (3, 2)\})$$

Dieser Graph G_1 hat drei Knoten, die Zahlen 1, 2 und 3. Er besitzt drei Kanten von der Zahl 1 zur Zahl 2 und von der Zahl 3 jeweils zu den Zahlen 1 und 2. Man kann Graphen auch graphisch darstellen, der Graph G_1 ist in Abbildung 1 visualisiert. Graphen wie diese haben in vielen Bereichen der Wissenschaft und Technik herausragende Bedeutung. Sie werden in der Elektrotechnik z.B. genutzt um Schaltungsnetze zu modellieren, in der Informatik um Prozesse und Datenstrukturen zu modellieren, in der Mathematik zur Modellierung diskreter Topologien; die Liste der Anwendungen kann beliebig weitergeführt werden. Praktische Relevanz im Alltag findet sich in der Straßennavigation, wo Orte durch Knoten und Verbindungswege durch Kanten modelliert werden können.

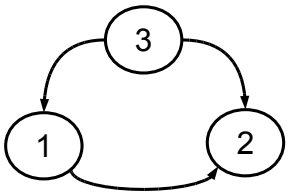

Abbildung 1: Grafische Darstellung des Graphen G_1

Bleiben wir kurz in der Straßennavigation, denn an dieser Anwendung lassen sich alle wichtigen Eigenschaften von Graphen gut erklären. Der in Abbildung 2 dargestellte Graph zeigt ein Modell der Verbindungswege zwischen den “Verkehrsknotenpunkten” Braunschweig, Hannover und Hamburg. Die Verbindungen zwischen den Knoten sind mit dem Namen der Autobahn beschriftet, welche die Knoten direkt verbindet. Mit diesem Graphen kann man “berechnen”, welchen Weg man nehmen kann, um von Braunschweig nach Hamburg zu gelangen. Wären in dem Graphen alle Stätte Deutschlands mit allen Straßen und ihren Namen eingetragen, so könnte man eine ganze Reihe von Wegen zwischen beliebigen Orten finden.

Allerdings kann es sein, dass der berechnete Weg nicht der kürzeste ist, denn in dem Graphen gibt es keine Daten über Distanzen zwischen den Knoten, welche allerdings erforderlich sind, um eine Minimierung der Gesamtwegstrecke durchzuführen. Diese Funktionalität kennen Sie von modernen Navigationsgeräten. Um den Informationsgehalt des Graphen zu erhöhen, kann man ihn mit weiteren Daten

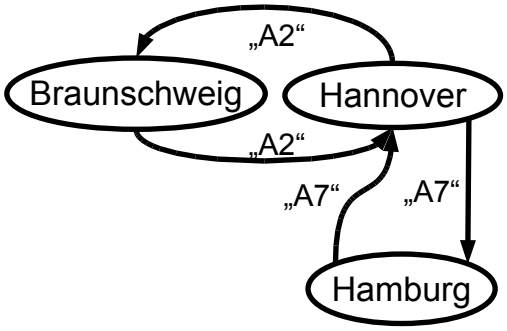

Abbildung 2: Eine vereinfachte Graphenrepräsentation der Verbindungswege zwischen Hamburg, Hannover und Braunschweig.

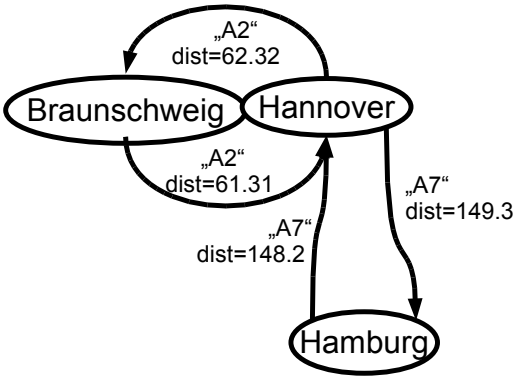

Abbildung 3: Eine vereinfachte Graphenrepräsentation der Verbindungswege zwischen Hamburg, Hannover und Braunschweig mit annotierten Distanzen.

anreichern, welche z.B. die Entfernung in Autobahnkilometern auf einer Kante oder die Stauwahrscheinlichkeiten beschreiben. Dies ist beispielhaft in Abbildung 3 geschehen. Wenn die Daten mit der realen Situation übereinstimmen, kann man mit einem solchen Graphen tatsächlich denen kürzesten Verkehrsweg zwischen zwei Verkehrsknotenpunkten ermitteln. Reichert man den Graphen dann weiter an, so dass die Koordinaten der Stätte zu den Datensätzen gehören (also weitere Informationen zu den Knoten einfließen), so kann man z.B. auch die geometrische Entfernung der Knotenpunkte berechnen. Sie sehen, dass die Menge der Anwendungen nur von der Fantasie begrenzt ist.

Eine Sprache zur Beschreibung von Graphen

Im letzten Abschnitt ist definiert worden, dass ein Graph formal aus einer Menge von Knoten (engl. Nodes) und einer Menge von Kanten (engl. Edges) besteht.

$$G = (V, E)$$

Um also eine Sprache zur Beschreibung von Graphen beschreiben zu können, muss man genau diese Elemente abbilden, also Kanten und Knoten definieren können. Ein Beispiel für eine Sprachdefinition in BNF ist:

```
<ID>                ::= <INT    '%i'>
<NAME>              ::= <STRING '%s'>
<REAL>              ::= <FLOAT  '%f'>

<NODEDEFINITION>    ::= NODE ( <NAME> )
<EDGEDEFINITION>    ::= EDGE ( <ID> , <ID> ,<REAL> )
```

Wie Sie sehen wurde in den ersten drei Ausdruckstypen auf eine formal vollständige Definition verzichtet. Die Funktion `scanf` hat bereits Interpreter für diese Ausdrücke eingebaut, wodurch wir eine Menge Arbeit sparen können. Die oben beschriebenen Ausdrücke können sogar vollständig von `scanf` interpretiert werden.

Übung 1: *Kürzeste Wege Algorithmen – Floyd-Warshall*

Zunächst sollten Sie sich intensiv mit dem von uns vorgegebenen Code auseinandersetzen. Wenn Sie die Struktur verstanden haben, werden Sie bemerken, dass das Programm sehr einfach erweitert werden

kann. Daher lassen Sie sich auf keinen Fall abschrecken und lassen Sie sich Zeit damit, den Code genau zu verstehen, dann fallen Ihnen die folgenden Aufgaben leichter.

```
/* DEFINING THE TYPES OF LEGAL EXPRESSIONS IN expressiontype.h */
typedef enum expression_type_enum
{  NODE=0, EDGE=1, HELP=2, SP=3 , END=254, ERROR=255 } expression_type;

//ALL STRUCTS ARE DEFINED IN struct.h
typedef struct edge_struct
{
  int source, target;
  float weight;
} Edge;

typedef struct node_struct
{
  char* name;
} Node;

typedef struct reader_context_struct
{
  Edge   edge;
  Node   node;
} Reader_Context;

expression_type readline(FILE* fp, Reader_Context* context)
{
  int string_length=0;
  char string[81],name[40];
  if(fgets(string,80,fp))
  {
    if(sscanf(string,"EDGE( %i , %i , %f )", &context->edge.source,
         &context->edge.target, &context->edge.weight) == 3
    )
      return EDGE;
    if(sscanf(string,"NODE( %39s )",name) == 1)
    {
      //strings are arrays of char and need to be copied manually...
      string_length=strlen(name);
      context->node.name=(char*)malloc(sizeof(char)*(name_length));
      memcpy(context->node.name,name,name_length);
      //the way scanf works it may read in the closing ')'
      if(context->node.name[name_length-1]==')')
        context->node.name[name_length-1]='\0';
      return NODE;
    }
    if(!strncmp(string,"HELP",4))
      return HELP;
    else return ERROR;
  }
  else return END;
```

```
49 }
```

Listing 1: `readline.c` mit eingebetteten Teilen aus `struct.h` und `expressiontype.h`

Listing 1 macht in den Zeilen 1–22 zunächst ein paar Typdefinitionen. Darunter wird in Zeile 2 zum ersten mal auf diesen Aufgabenblättern ein Aufzählungstyp (`enum`) benutzt. Bitte lesen Sie dazu im Buch Buch [Dausmann,Bröckl,Goll:2008] das entsprechende Kapitel. Die Zeilen 23–49 zeigen den Code einer Funktion, welche die beiden oben formal definierten Ausdrücke `NODE(...)` und `EDGE(...)`, einliest, und zusätzlich in Zeile 44 die Eingabe auf Gleichheit mit der Zeichenfolge “HELP” überprüft. Schauen Sie sich genau an, was dieser Code macht und wie er es macht. Die Funktion liest sich wie folgt:

```
expression_type readline(source_file, data)
{
  char string[81], name[40];
  if reading line from source_file is possible, read it to string
  {
    if line can be interpreted as EDGEDEFINITION, read in edge to data
      return "read an edge";
    if line...
  }
  else file must be at end, so return "FILE AT END"
}
```

Die Funktion gibt also eine Zahl zurück, welche den Typ des eingelesene Ausdrucks angibt. Die Daten, die der eingelesene Ausdruck definiert werden im `context` abgelegt.
Diese Funktion wird in Listing 2 benutzt, um einen Graphen aus einer Datei einzulesen (Dieses Programm ist eine leicht verkürzte Fassung die nur von `stdin` und nicht aus beliebigen Dateien liest). Der Aufruf der Funktion `readline` findet in Zeile 10 statt. Die in `readline` eingelesenen Daten werden dann in der Funktion `interpret` in die Datenfelder `edge_array` und `node_array` übertragen.

```
1  #define MAX_NODE 2000
2  #define MAX_EDGE 2000
3  int main(int argc, char** argv)
4  {
5    int            type,num_edge=0,num_node=0;
6    Edge           edge_array[MAX_EDGE];
7    Node           node_array[MAX_NODE];
8    Reader_Context context;
9
10   while((type=readline(stdin,&context))!=END)
11   {
12     interpret(stderr,context,type,&num_edge,edge_array,&num_node,node_array);
13   }
14
15   printf("\n");
16   return 0;
17 }
```

Listing 2: main.c

Die Funktion `interpret` ist in leicht vereinfachter Form in Listing 3 gegeben. Die im Kontext befindlichen Informationen werden hier unter der Verwendung des identifizierten Ausdruckstypen in die entsprechende Datenfelder übertragen.

```
void interpret(FILE* out,Reader_Context context, expression_type type,
               int* num_edge, Edge* edge_array,
               int* num_node, Node* node_array)
{
  switch(type)
  {
    case NODE:
      node_array[*num_node]=context.node;
      (*num_node)++;
      return;
    case EDGE:
      edge_array[*num_edge]=context.edge;
      (*num_edge)++;
      return;
    case HELP:
      print_help(out);
      return;
    case ERROR:
      fprintf(out,"\nERROR READING DATA\n");
      return;
    default:
      return;
  }
}
```

Listing 3: Die Funktion `interpret`

Mit dem gerade vorgestellten Programm ist es möglich, einen Graphen aus einer Datei einzulesen um auf ihm Algorithmen auszuführen. Ein wichtiges Problem vor das einen ein komplexer Graph stellt ist das finden des kürzesten Weges zwischen zwei beliebigen Knoten. Bei dem Beispiel in Abbildung 3 ist es noch relativ einfach den kürzesten Weg zwischen zwei Knoten zu finden, denn es gibt keine Alternativwege. Ist der Graph allerdings größer und gibt es mehrere verschiedene Wege, so ist dieses Problem nicht mehr so leicht zu lösen. Im schlimmsten Fall muss man systematisch jeden denkbaren Weg durchgehen und den kürzesten suchen.
Dieses Vorgehen setzt der sogenannte Floyd-Warshall Algorithmus um, den Sie in dieser Aufgabe implementieren sollen. Dieser Algorithmus würde in einem modernen Navigationssystem zwar keine Anwendung mehr finden, dennoch ist er gerade für den Anfang und für kleine Graphen gut geeignet. Das große Problem dieses Verfahrens ist, dass seine Rechenkomplexität kubisch von der Anzahl der Knoten abhängt. Das ist mit einem schnellen Rechner und relativ wenigen Knoten natürlich nicht das größte Problem, aber zusätzlich benötigt der Algorithmus eine $n \times n$ Matrix, welche bei einer Knotenzahl von 1.000 bereits eine größe von 1.000.000 annimmt und somit bei Benutzung von `double` Variablen (die 8 Byte Speicher einnehmen), bereits 8 MegaByte Speicher und mit 10.000 Knoten schon 100.000.000 schon 800 MegaByte einnimmt. Was nützt ein Navigationssystem, das mit gerade mal 10.000 Kreuzungen schon einen relativ großen Computer benötigt?
Lässt man dieses Problem außer Acht, so ist der Floyd-Warshall Algorithmus ein relative einfacher und damit gut verständlicher Algorithmus zur Lösung des Problems der kürzesten Wege.

Bemerkung: Der Floyd-Warshall Algorithmus wird hier als Pseudocode präsentiert.

```
    //Assume a function edge_cost(i,j,..) returns the cost of the edge
    //from i to j (infinity if there is none) and edge_cost(i,i,...)=0.

    //Assume path is an n × n Matrix, where n is the number of nodes
 1  FLOYDWARSHALL(source_id, target_id, n, m, edges)
    //source_id is the id of the source node
    //target_id is the id of the target node
    //n is the number of nodes
    //m is the number of edges
    //edges is a data structure holding the edges of the graph

 2  Matrix path[n][n] //n × n-Matrix
    //initialisation phase
 3  for i ← 1 to n
 4      do for j ← 1 to n
 5          do path[i][j] = edge_cost (i,j,m,edges)
    //floyd-warshall phase
 6  for k ← 1 to n
 7      do for i ← 1 to n
 8          do for j ← 1 to n
 9              do path[i][j] = MIN( path[i][j], path[i][k]+path[k][j] )
10  return ←path[source_id][target_id]
```

Aufgabe a) Implementieren Sie den Floyd Warshall Algorithmus in der Datei `floydwarshall.c`, welche im Paket `http://www.wire.tu-bs.de/ADV/files/graph_processor/graph_processor_template.zip` enthalten ist. Wie Sie sehen haben wir Ihnen die Arbeit zur Implementierung der Funktion `edge_cost` abgenommen, damit es für Sie etwas leichter wird. □

Aufgabe b) Bauen Sie einen neuen Befehl `SHORTESTPATH(#i,#j)` in den Sprachinterpreter ein (dazu müssen nur die Dateien `interpret.c` und `readline.c` erweitert werden.). Die BNF zu dem neuen Befehl lautet:

```
<SHORTESTPATHCALL>    ::= SHORTESTPATH ( <ID> , <ID> )
```

□

Aufgabe c) Starten Sie das Programm mit dem Aufruf `./main data/graph.in`, welcher zunächst den Graphen aus der Dateien `data/graph.in` einliest. Ein paar Beispiele für kürzeste Wege, die unsere Implementierung des Floyd-Warshall Algorithmus berechnet lauten:

```
SHORTESTPATH(Hamburg--(210.52)-->Braunschweig)
SHORTESTPATH(Hamburg--(328.3)-->Münster)
```

`SHORTESTPATH(Münster--(494.52)-->Berlin)`

Testen Sie ihr Programm mit weiteren Eingaben. □

Aufgabe d) Geben Sie weitere Knoten und Kanten in die Datei `data/graph.in` ein und testen Sie den Algorithmus weiter. □

Bis jetzt haben wir uns damit beschäftigt, wie man komplexe Datenstrukturen abspeichern und wieder einlesen kann. Diese Aufgabe muss in der Praxis der Programmentwicklung sehr häufig bewältigt werden. Auch dynamische Datenstrukturen sind in der Praxis ausgesprochen wichtig, da man ohne Sie nur sehr unflexible Programme schreiben kann, und bestimmte Algorithmen nicht implementiert werden können. Wir haben auf den letzten Aufgabenblättern bereits Such- und Sortierverfahren betrachtet, welche ein Array als Basisstruktur benutzen. Viele Algorithmen benötigen allerdings Strukturen wie Listen, bei denen Einträge auch in der Mitte eingefügt werden können (das geht bei Arrays nicht) oder Bäume, auf denen man noch schneller als in log_2 n Schritten suchen kann[1]. Da dynamische Datenstrukturen also eine äußerst wichtige Rolle spielen, werden wir in dieser Aufgabe auf diese Technik eingehen und eine *doppelt verkettete Liste* umsetzen.
Die in Abbildung 4 dargestellte Liste besteht technisch aus zwei Strukturen (structs).

`struct List` Die Hauptstruktur `List`, die im dunkleren Bereich des Bildes 4 dargestellt ist, speichert zwei Zeiger auf Listeneinträge, die auf den Kopf (`head`) und auf das Ende (`tail`) der Liste zeigen. Der dritte Zeiger dient dazu, auf ein bestimmtes Element der Liste zu zeigen, er ist wie ein Zeigefinger, mit dem man eine Liste durchsucht. Dieser Zeigefinger wird im technischen Sprachgebrauch als Iterator (kurz `iter`) bezeichnet.

`struct List_Entry` Eine doppelt verkettete Liste ist eine Liste in der man an jedem Eintrag sehen kann, welcher der vorige (previous — `prev`) und welcher der nächste Eintrag in der Liste (next — `next`) ist. Da jeder Eintrag auch einen Wert besitzt, besitzt die Struktur `List_Entry` eine Variable, die den Wert des Listeneintrags enthält. In der Liste aus Abbildung 4 kann man z.B. mit dem Ausdruck `List.head->next->value` auf den Wert des zweiten Eintrags der Liste zugreifen (falls er existiert). Im Listing 4 werden diese Strukturen in C-Syntax definiert, hierbei kann man sehr schön die Verbindung zwischen Abbildung 4 und ihrer Umsetzung in C erkennen.

[1]Nehmen Sie die Datenbank aller Autos die in Deutschland gemeldet sind. Nehmen Sie an, die Identifikationsnummern sind sortiert, so dass man in log_2 n Schritten jeden Datensatz findet. Jetzt sollen aber alle Autos gefunden werden, die Weiß sind, da hilft diese Suche nicht mehr und die Datenbank muss linear in O(n) Schritten durchsucht werden. Nehmen wir an ein Datensatz hat 1.000 Byte, bei 50.000.000 Fahrzeugen in Deutschland hat die Datenbank also ca. 50GB an Größe. Auf einer handelsüblichen Festplatte dauert die Suche (bei maximal 50MB/s) somit noch mindestens eine $\frac{1}{4}$ Stunde. Datenbanken sind ein wichtiges Anwendungsgebiet von dynamischen Datenstrukturen.

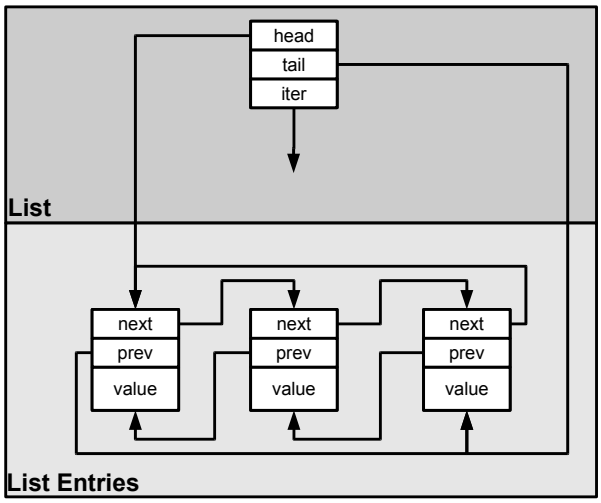

Abbildung 4: Graphische Darstellung einer Liste, wie Sie in C implementiert werden kann.

```
1  typedef struct List_entry_struct
2  {
3     struct List_entry_struct* next;
4     struct List_entry_struct* prev;
5     value_type    value;
6  } List_entry;
7
8  typedef struct List_struct
9  {
10    List_entry* head;
11    List_entry* tail;
12    List_entry* iter;
13    int         size;
14 }  List;
```

Listing 4: Technische Umsetzung der in Abbildung 4 dargestellten Liste.

Bemerkung: Die in Listing 4 benutzte Schreibweise zur Definition einer Struktur hat folgenden Grund:

```
1 struct demo
2 {
3   int demo_value;
4 };
5 struct demo instance;
```

Listing 5: Strukturdefinitionen

Die in Listing 5 definierte Struktur `demo` kann im Programmcode nur zusammen mit dem Wort `struct` verwendet werden, da eine Struktur in C nicht automatisch ein Typ ist, sondern eben eine Struktur.

```
1 struct demo
2 {
3   int demo_value;
4 };
5 typedef struct demo demoType;
6 demoType instance;
```

Listing 6: Strukturdefinitionen mit Typdefinition

Die in Listing 6 definierte Struktur `demo` ist identisch mit der Definition aus Listing 5. Zusätzlich wurde die Struktur in Zeile 5 explizit zu einem Typen `demoType` gemacht. Dieser Typ wird dann in Zeile 6 ganz normal als Typ benutzt.

```
1 typedef struct demo_struct
2 {
3   int demo_value;
4 } demo;
5 demo instance;
```

Listing 7: Kurzform der Deklaration eines Strukturtypen

Die in Listing 7 verwendete Notation ist kompakter, macht aber eigentlich das Gleiche. Hier wird ein Typ `demo` definiert, der ein `struct demo_struct` *ist*. Der Bezeichner `demo` ist im folgenden Code synonym for `struct demo_struct` zu verwenden.

Prinzip der Trennung von Algorithmen und Datenstrukturen

Wenn man ein Programm schreiben möchte, das eine Liste benutzt, hat man zwei Möglichkeiten dies zu tun:

Eingebettet die Operationen zur Manipulation einer Liste werden *direkt in den Code* integriert, ein neuer Eintrag erfordert dann speziellen Code innerhalb eines Algorithmus der den neuen Eintrag in die Liste einfügt, also die Zeiger (`next`,`prev`, ...) entsprechend verändert.

Gekapselt die Operationen zur Manipulation einer Liste werden *separat in Form von Funktionen* definiert. Es gibt also Funktionen wie `insert, append, ...` welche eine gegebene Liste manipulieren.

Die Einbettung der Manipulationsoperatoren führt dazu, dass technischer Programmcode und Anwendungscode miteinander stark verwoben sind. Diese Verwebung ergibt, dass das resultierende Programm nicht mehr so leicht verändert werden kann. Stellen Sie sich vor, dass die Listenimplementierung, die ein Programm benutzt, verändert werden soll und Elemente beispielsweise auf eine andere Art in die Liste eingebunden werden sollen. In der eingebetteten Variante muss der gesamte Programmcode durchsucht werden und an vielen Stellen angepasst werden. In der gekapselten Variante beschränkt sich die Codeänderung auf einen kleinen Satz von Funktionen. Damit ist klar, eine Verwebung von Datenstrukturen und Anwendungsalgorithmen hat schwerwiegende Nachteile.
In dieser Aufgabe soll die vorteilhafte Separierung von Algorithmen und Datenstrukturen am Beispiel einer Liste geübt werden. Dazu werden wir wieder eine Header-Datei vorgeben, welche die zu implementierenden Funktionen deklariert. Die Liste soll in einer späteren Aufgabe dazu benutzt werden, Routen zwischen Städten zu beschreiben; sie soll also dazu dienen, die Funktionalität eines Navigationssystems nachzubilden. Aus diesem Grund wird als Wertetyp (`value_type`) eines Listeneintrags der Typ `int` angenommen, welcher später die Knotenidentifikationsnummern einer Route enthalten soll. Als Headerdatei ergibt sich daher Listing 8.

```
1  typedef enum boolean_enum {TRUE=1, FALSE=0} bool;
2  typedef struct Int_List_entry_struct
3  {
4     int                          value;
5     struct Int_List_entry_struct* next;
6     struct Int_List_entry_struct* prev;
7  } Int_List_entry;
8
9  typedef struct int_List_struct
10 {
11    Int_List_entry* head;
12    Int_List_entry* tail;
13    Int_List_entry* iter;
14    int             size;
15 } Int_List;
16
17 /* Basic List Operations */
18   /* create and destroy */
```

```
19      /* setting Int_List entries to NULL */
20      void init_int_list(Int_List*);
21      /* delete all entries and re-init list */
22      void free_int_list(Int_List*);
23
24    /* state request */
25      /* TRUE if list is empty */
26      bool empty_int_list(Int_List*);
27      /* returns size of given list */
28      int  size_int_list(Int_List*);
29
30    /* append and prepend entries */
31      /* appends value to list */
32      void append_int_list(Int_List*,int);
33      /* prepends value to list */
34      void prepend_int_list(Int_List*,int);
35
36    /* insert list into list */
37      /* inserts second list into the first behind position of iter */
38      /* second list is empty after this operation*/
39      void int_list_insert_at_iter(Int_List*,Int_List*);
40
41  /* ITERATOR FUNCTIONS */
42    /* move iter to tail or head of list*/
43      void tail_int_list_iter(Int_List*);
44      void head_int_list_iter(Int_List*);
45
46    /* reading and writing value at position of iter */
47      int  read_int_list_iter(Int_List*);
48      void write_int_list_iter(Int_List*, int);
49
50    /* increment/decrement iterator (move next/prev) */
51    /* if decrement or increment at head or tail respectively */
52    /* the iterator points to NULL */
53      void incr_int_list_iter(Int_List*);
54      void decr_int_list_iter(Int_List*);
55
56    /* state request of iterator */
57      /* return iter==tail */
58      bool is_tail_int_list_iter(Int_List*);
59      /* return iter==head */
60      bool is_head_int_list_iter(Int_List*);
61      /* return iter==NULL */
62      bool is_NULL_int_list_iter(Int_List*);
```

Listing 8: int_list.h

Da in Listing 8 alle Funktionen bereits deklariert sind, die für die Umsetzung einer Liste erforderlich sind, können wir Ihnen wieder einen Test vorgeben, der das korrekte Verhalten der Liste überprüft. Dieser Test ist in Listing 9 gegeben.

```
1 #include "int_list.h"
```

```c
#include <assert.h>
#include <stdio.h>
int main()
{
  int i;
  Int_List list1;
  Int_List list2;
  init_int_list(&list1);
  init_int_list(&list2);
  for(i=0;i<10;i++)
  {
    append_int_list(&list1,i);
  }
  assert(size_int_list(&list1)==10);
  printf("MARK1\n");
  /* testing for loop with use of iterator*/
  for( head_int_list_iter(&list1),i=0;
       !is_NULL_int_list_iter(&list1);
       incr_int_list_iter(&list1),i++)
  {
    assert(read_int_list_iter(&list1)==i);
    printf("List[%i]==%i\n",i,read_int_list_iter(&list1));
  }
  free_int_list(&list1);
  for(i=0;i<10;i++)
  {
    append_int_list(&list1,i);
    append_int_list(&list2,10+i);
  }
  tail_int_list_iter(&list1);
  int_list_insert_at_iter(&list1,&list2);
  assert(size_int_list(&list2)==0);
  assert(size_int_list(&list1)==20);
  printf("MARK2\n");
  for(head_int_list_iter(&list1),i=0;
       !is_NULL_int_list_iter(&list1);
       incr_int_list_iter(&list1),i++ )
  {
    assert(read_int_list_iter(&list1)==i);
    printf("List[%i]==%i\n",i,read_int_list_iter(&list1));
  }
  printf("%i\n",i);
  assert(i==20);
  free_int_list(&list1);
  free_int_list(&list2);
  return 0;
}
```

Listing 9: int_list_test.c

Übung 2: *Listenimplementierung*

Aufgabe a) Laden Sie die vorgegebenen Dateien unter `http://www.wire.tu-bs.de/ADV/files/List/` herunter. □

Aufgabe b) Implementieren Sie die in der Datei `int_list.h` deklarierten Funktionen in einer Datei `int_list.c` und schreiben Sie ein Makefile, das den Test mit Ihrer Implementierung verbindet. Damit Sie in den folgenden Aufgaben keine Probleme bekommen, sollte der Test ohne Fehler durchlaufen. □

Aufgabe c) Ergänzen Sie ihr Makefile, so dass ihre Listenimplementierung auch in einer Bibliothek (`libintlist.a`) verpackt wird. □

Anwendung der `Int_List` bei der Auswertung von kürzesten Routen

In der letzten Aufgabe wurde der Floyd-Warschall Algorithmus benutzt, um die kürzeste Entfernung zwischen zwei Verkehrsknotenpunkten zu berechnen. Hier werden wir den Algorithmus etwas erweitern und zusätzlich den Weg selbst, also eine Route, berechnen.
In dieser Anwendung berechnet der Floyd-Warschall Algorithmus eine Matrix $A \in \mathbb{N}^{n \times n}$ (mit n gleich der Anzahl an Knoten im Graphen), in der für je zwei Knoten i und j ein Eintrag $A_{i,j} = k$ berechnet wird, der einen Zwischenknoten angibt, über den der kürzeste Weg zwischen i und j führt. Da Knoten in unserem Programm über ganzzahlige Werte (`int`) identifiziert werden, wird für diese Aufgabe (wie schon auf dem zweiten Aufgabenblatt) erneut eine `square_matrix`, diesmal allerdings mit `int`-Werten benötigt.
Übung 3: *Matrix für ganze Zahlen*
Listing 10 gibt eine Erweiterung der Datei `lin_struct.h` vom zweiten Aufgabenblatt an, welche nochmals umgesetzt werden soll. Dazu sollten Sie nicht mehr als 10 Minuten benötigen, denn Sie können den alten Code kopieren.

```
#ifndef LINSTRUCT__H
#define LINSTRUCT__H
  double* create_double_vector(int size);
  void free_double_vector(double*);
  double** create_double_square_matrix(int size);
  void free_double_square_matrix(double**,int size);

  int* create_int_vector(int size);
  void free_int_vector(int*);
  int** create_int_square_matrix(int size);
  void free_int_square_matrix(int**,int size);
#endif
```

Listing 10: lin_struct.h

Aufgabe a) Verändern Sie die Implementierung aus der ersten Übung vom zweiten Aufgabenblatt, so dass die in Listing 10 deklarierten Funktionen implementiert werden. □

Das folgende Listing 11 macht zusätzliche Funktionsdeklarationen welche in Datei `floydwarshall.h` eingefügt und in `floydwarshall.c` implementiert werden sollen.

```
#ifndef __FLOYDWARSHALL__H
#define __FLOYDWARSHALL__H
#include "struct.h"
#include <float.h>
#include <stdlib.h>
#include <assert.h>
#include <stdio.h>

  ...

/*****************************************************************************/
/* function edge_target: gives the target of a given edge                    */
/*   if the edge is defined: b is returned                                   */
/*   if the edge does not exist: -1 is returned                              */
/*   if a and b are the same node: b is returned                             */
/* args: a(id of source node) b(id of target node)                           */
/*       num_edge(number of edges in given array of edges)                   */
/*       edge(array of edges representing the graph)                         */
/*****************************************************************************/
  int edge_target(int a, int b, int num_edge, Edge* edge);

/*****************************************************************************/
/* function evaluate_optimal_route: returns the optimal route between        */
/*    source and destination recursively                                     */
/* args: path(matrix of optimal way points)                                  */
/*       source(id of source node), target(id of target node)                */
/* return: List of waypoints between source and destination                  */
/*****************************************************************************/
  Int_List evaluate_optimal_route(int** route, int source, int dest);

/*****************************************************************************/
/* function floydwarshall_route: compute the shortest path from              */
/*   request->source node to request->target node through a graph given      */
/*   by an array of edges (parameter Edge* edge); the Edge struct is         */
/*   defined in file struct.h.                                               */
/* args:   request(node with requested source and target information),       */
/*         route(return a list of all waypoints),                            */
/*         num_node(number of nodes in graph), edge(edges of the graph)      */
/* return: int point value(a value of type floyd_warshall_status)            */
/*         SUCCESS on success; FEW_NODES of requested nodes are not in       */
/*         graph request->weight contains the estimated shortest distance */
/*****************************************************************************/
  int floydwarshall_route(Edge* request,Int_List* waypoints, int num_node,
                          int num_edge, Edge* edge);

  ...

```

```
48 #endif
```

Listing 11: floydwarshall.h

Bemerkung: Der Floyd-Warshall Algorithmus mit Berechnung des kürzesten Pfad wird hier als Pseudocode präsentiert.

```
   //Assume a function edge_cost(i,j,..) returns the cost of the edge
   //from i to j (infinity if there is none) and edge_cost(i,i,...)=0.
   //Assume a function edge_target(i,j,..) returns the target of the edge
   //from i to j (-1 if there is none) and edge_target(i,j,...)=j.

   //Assume path_cost is a n × n Matrix, where n is the number of nodes
   //Assume route is a n × n Matrix, where n is the number of nodes
1  FLOYDWARSHALLROUTE(source_id, target_id, n, m, edges)
   //source_id is the id of the source node
   //target_id is the id of the target node
   //n is the number of nodes
   //m is the number of edges
   //edges is a data structure holding the edges of the graph

2  Matrix path_cost [n][n] //n × n-Matrix of type double
3  Matrix route[n][n] //n × n-Matrix of type int
   //initialisation phase
4  for i ← 1 to n
5       do for j ← 1 to n
6            do path_cost [i][j] = edge_cost (i,j,m,edges)
7               route[i][j] = edge_target (i,j,m,edges)
   //floyd-warshall phase
8  for k ← 1 to n
9       do for i ← 1 to n
10           do for j ← 1 to n
11                do if ( path_cost[i][j] ¿ path_cost[i][k]+path_cost[k][j] )
12                     do path_cost[i][j] = path_cost[i][k]+path_cost[k][j]
13                        route[i][j] = k
14 waypoints ← EVALUATE_OPTIMAL_ROUTE(route, source_id, target_id)
15 return ←path_cost[source_id][target_id]
```

Die im Pseudocode gegebenen Zeilen 14 und 15 sollen beide als Rückgabewert interpretiert werden. Die Variable `waypoints` ist wie in Listing 11 in Zeile 43 gegeben, als zweiter Rückgabewert zu betrachten (es wird ein Zeiger übergeben).

Bemerkung: Im folgenden geben wir den Pseudocode für die Funktion `evaluate_optimal_route` an.

```
    //Assume route is a n × n Matrix, where n is the number of nodes
 1  EVALUATE_OPTIMAL_ROUTE(route, source_id, target_id)
    //route is the route matrix computed by FloydWarshallRoute
    //source_id is the id of the source node
    //target_id is the id of the target node
    //return is the return value of the Function of type Int_List
 2  int waypoint=path[source_id][target_id]
 3  if (source_id!=target_id && target_id!=path[source_id][target_id])
 4      do //subroute is of type Int_List
 5          subroute = EVALUATE_OPTIMAL_ROUTE(route, source_id, waypoint)
 6          TAIL_INT_LIST_ITER(return)
 7          INT_LIST_INSERT_AT_ITER(return, subroute)
 8          subroute = EVALUATE_OPTIMAL_ROUTE(route, waypoint, target_id)
 9          TAIL_INT_LIST_ITER(return)
10          INT_LIST_INSERT_AT_ITER(return, subroute)
11    else  APPEND_INT_LIST(return, waypoint)
```

Übung 4: *Der kürzeste Weg*

Nun haben wir endgültig alle Werkzeuge zusammen, um den kürzesten Weg zwischen zwei Knoten in einem Graphen zu berechnen.

Aufgabe a) Implementieren Sie die Funktionen aus Listing 11 in der Datei `floydwarshall.c`. Benutzen Sie dabei als `Int_List` die von Ihnen auf diesem Aufgabenblatt implementierte Listenimplementierung.

Bemerkung: Achten Sie bei der Implementierung darauf, dass sie allokierten Speicher auch wieder frei geben, wenn Sie ihn nicht mehr benötigen. Im Pseudocode stehen weder Initialisierungen noch Löschoperationen für die aufgestellten Matrizen und Listen.

□

Aufgabe b) Bauen Sie einen neuen Befehl `SHORTESTROUTE(#i,#j)` in den vorhandenen Sprachinterpreter ein (dazu müssen nur die Dateien `interpret.c` und `readline.c` erweitert werden.). Die BNF zu dem neuen Befehl lautet:

```
<SHORTESTROUTECALL>    ::= SHORTESTROUTE ( <ID> , <ID> )
```

Achten Sie darauf, dass in der Ausgabe zu dem `SHORTESTROUTE`-Befehl der Weg zwischen den beiden Knoten sinnvoll ausgegeben wird. Ein Beispiel für eine sinnvolle Ausgabe sehen Sie in Listing 12.

```
1 SHORTESTROUTE(Hamburg--(210.52)-->Braunschweig)
2       ROUTE-STAGE(Hamburg--(148.2)-->Hannover)
3       ROUTE-STAGE(Hannover--(62.32)-->Braunschweig)
4 SHORTESTROUTE(Hamburg--(293.7)-->Muenster)
5       ROUTE-STAGE(Hamburg--(123.9)-->Bremen)
6       ROUTE-STAGE(Bremen--(169.8)-->Muenster)
```

```
SHORTESTROUTE(Muenster--(494.52)-->Berlin)
     ROUTE-STAGE(Muenster--(178.2)-->Hannover)
     ROUTE-STAGE(Hannover--(62.32)-->Braunschweig)
     ROUTE-STAGE(Braunschweig--(96.8)-->Magdeburg)
     ROUTE-STAGE(Magdeburg--(157.2)-->Berlin)
```

Listing 12: Beispielhafte Ausgabe des Programms

□

Bemerkung: Die Eingabedatei zu der Ausgabe in Listing 12 finden Sie unter `http://www.wire.tu-bs.de/ADV/files/graph/`.