Aufgabe 1. Schreibe ein Programm, dass ein Labyrinth aus einer Datei einliest:

```
XXXXXXXXXXXXXXXX
X X XXXXXXXXXX*X
X$X XX     XXX X
X X XX XXX XXX X
X   XX XXX XXX X
XXX X   XX XXX X
XXX   X        X
XXXXXXXXXXXXXXXX
```

Bemerkung: Wir spezifizieren das Labyrinth hier nicht viel näher, entscheide dich selbst vorher für ein Format. Soll die Größe des Labyrinths variabel sein oder fest? Soll die Größe in der ersten Zeile der Datei stehen oder nicht? Soll das Labyrinth quadratisch sein oder nicht? Soll es außen herum immer mit Xen begrenzt sein oder hast du vielleicht eine andere Lösung?

Das Programm soll einen Weg vom Startpunkt (dem Stern) zum Schatz (dem Dollarzeichen) finden. Die Xe sind Wände und Leerzeichen sind Pfade. Markiere einen Weg mit Punkten und gebe das Labyrinth mit Weg in der Konsole aus.

```
XXXXXXXXXXXXXXXX
X X XXXXXXXXXX*X
X$X XX     XXX.X
X.X XX XXX XXX.X
X...XX XXX XXX.X
XXX.X...XX XXX.X
XXX...X........X
XXXXXXXXXXXXXXXX
```

Aufgabe 2. Implementiere ein Modul, das Rechenoperationen für ganzzahlige Brüche bereitstellt:

```
1 typedef struct {
2       signed long int numerator;
3     unsigned long int denominator;
4 } RATIONAL;
```

Es sollte Funktionen zum addieren, subtrahieren und multiplizieren von Brüchen geben. Das Ergebnis einer Rechnung sollte immer vollständig gekürzt sein.

Aufgabe 3. Lies im Skript den Teil 7.5 über verkettete Listen. Implementiere doppelt verkettete Listen, die anstatt einer `double`-Variable beliebige Daten speichern können, als `void*`. Die Header-Datei könnte etwa wie folgt aussehen:

```
1  #ifndef _LIST__H
2  #define _LIST__H
3
4  LIST *list_create(); /* Leere Liste erstellen */
5
6  /* Element hinter E einfügen, NULL heißt am Anfang */
7  LISTNODE *list_insert(LIST *L, LISTNODE *E, void *p);
8
9  /* Element am Anfang bzw. Ende einfügen */
10 LISTNODE *list_unshift(LIST *L, void *p);
11 LISTNODE *list_push(LIST *L, void *p);
12
13 /* Element am Anfang bzw. Ende entfernen und
14    die Daten zurück geben */
15 void *list_shift(LIST *L);
16 void *list_pop(LIST *L);
17
18 /* eine Element aus der Liste entfernen */
19 void list_delete(LIST *L, LISTNODE *E);
20
21 /* Die Elemente aus Liste M an Liste L anhängen und
22    L zurückgeben. Nach Aufruf ist M leer. */
23 LIST *list_merge(LIST *L, LIST *M);
24
25 /* Liste inklusive allen Elementen frei geben */
26 void list_free(LIST *L);
27
28 #endif
```