3 C-Syntax II

Aufgabe 3.1. (verlinkte Listen – leicht) Schreiben Sie ein Programm, dass Zahlen einließt, bis `-1` eingegeben wird. Jede Zahl soll als Glied in eine verlinkte Liste eingehängt werden. Ist die Eingabe abgeschlossen, sollen die Zahlen in umgekehrter Reihenfolge ausgegeben werden. Allozieren Sie die Glieder der Liste mit **malloc**(3). Sie können folgenden Coderahmen für die Struktur der Liste verwenden.

```
/* Glied einer verlinkten Liste */
struct glied {
    int wert;
    struct glied *nachfolger;
};
```

Aufgabe 3.2. (Listen sortieren – mittel) Modifizieren Sie das Programm aus der vorherigen Aufgabe, sodass es die eingelesenen Zahlen sortiert ausgibt. Sortieren Sie dazu die von Ihrem Programm angelegte Liste mit dem Algorithmus *Mergesort*.

Aufgabe 3.3. (Einheitenumrechner – leicht) Schreiben Sie ein interaktives Programm, mit dem der Benutzer zwischen verschiedenen Einheiten umrechnen kann. Der Benutzer soll die Möglichkeit haben, zwei Einheiten und einen Betrag einzugeben, dass Programm gibt den Betrag dann in der Zieleinheit aus.

Denken Sie darüber nach, wie man das Programm gestalten kann, sodass eine Erweiterung auf zusätzliche Einheiten möglichst einfach ist. Hierzu ist es ggf. hilfreich, im Programm eine Tabelle anzulegen, die für jede unterstützte Einheit die Art der Einheit (z. B. Währung, Länge, Masse) und ihren Umrechnungsfaktor speichert.

Aufgabe 3.4. (Stringsuche – mittel) Schreiben Sie Ihre eigene Variante der Funktion `strstr`(3). Schreiben Sie ein Programm, um diese zu testen. Hierzu bietet es sich an, dass Ihre eigene Implementierung mit der von System bereitgestellten Implementierung zu vergleich.

Analysieren Sie die Laufzeit Ihrer Implementierung und recherchieren Sie bessere Algorithmen zur Suche von Strings in Strings.

Aufgabe 3.5. (Tabellen formatieren – mittel) Schreiben Sie ein Programm, das eine CSV-Datei einließt und formatiert als Tabelle ausgibt. Eine CSV-Datei ist eine Textdatei, die in jeder Zeile eine mit Kommata getrennte Liste von Feldern enthält. Die Inhalte der Felder können in Anführungszeichen gesetzt werden, wenn sie selbst Kommata enthalten, das müssen Sie aber nicht unterstützen.

```
Stadt,Einwohner,Flaeche,Land
Berlin,3613495,891.68,Berlin
Hamburg,1830584,755.22,Hamburg
Muenchen,1456039,310.70,Bayern
Koeln,1080394,405.02,Nordrhein-Westfalen
Frankfurt,746878,248.31,Hessen
Stuttgart,632743,207.35,Baden-Wuerttemberg
```

Obige Datei sollte formatiert etwa so aussehen:

```
Stadt       Einwohner  Flaeche  Land
Berlin      3613495    891.68   Berlin
Hamburg     1830584    755.22   Hamburg
Muenchen    1456039    310.70   Bayern
Koeln       1080394    405.02   Nordrhein-Westfalen
Frankfurt   746878     248.31   Hessen
Stuttgart   632743     207.35   Baden-Wuerttemberg
```

Versuchen Sie, ihr Programm so zu schreiben, dass es die Breite der Spalten an Hand des Inhaltes passend auswählt. Dazu empfiehlt es sich, den Inhalt der Datei zunächst einmal in den Arbeitsspeicher einzulesen. Im ersten Durchlauf können die nötigen Spaltenbreiten bestimmt werden, um die Spalten dann im zweiten Durchlauf optimal auszugeben.

Als Erweiterung können Sie mit den Zeichen `+`, `-`, und `|` Linien um die Zellen der Tabellen ziehen. Auch bietet es sich an, Zellen, die nur Zahlen enthalten in der Tabelle rechts statt links auszurichten.

Aufgabe 3.6. (`env`(1) – mittel) Schreiben Sie eine Implementierung des Programmes `env`(1). Dieses hat zwei Aufgaben:

Ohne Argumente aufgerufen, gibt `env`(1) den Inhalt der zur Zeit gesetzten Umgebungsvariablen aus. Diese können Sie durch Inspektion der globalen Variable `environ`(7) auslesen.

Werden Argumente übergeben, so wird jedes Argument der Form

`Schluessel=Wert`

mit `putenv`(3) als Variable in die Umgebung aufgenommen. Alle Argumente ab dem ersten Argument, das nicht diese Form hat, werden als Name und Argumente eines auszuführenden Programmes verstanden, welches nach Setzen der Umgebungsvariablen mit `execvp`(2) ausgeführt wird.

Weitere Funktionalität von `env`(1) ist nicht Teil der Aufgabe.

Aufgabe 3.7. (Worthäufigkeit – schwer) Schreiben Sie ein Programm, das Text aus der Eingabe ließt und zählt, wie oft jedes Wort in diesem vorkommt. Ist die Eingabe erschöpft, so geben Sie alle gefundenen Worte und ihre Häufigkeiten möglichst nach Häufigkeit sortiert aus. Achten Sie darauf, dass Satz- und Leerzeichen nicht Teil von Worten sind.

Machen Sie sinnvolle Annahmen über die Eingabe, z. B. dass kein Wort länger als 16 Buchstaben ist und dass es insgesamt nicht mehr als 1000 verschiedene Worte gibt. Denken Sie über eine geeignete Datenstruktur zur Ablage der Worte. Sie können zu diesem Zweck den oben angebeben Code-Rahmen um weitere Felder und Strukturen erweitern.

Aufgabe 3.8. (base64 – schwer) Schreiben Sie zwei Programme, die Dateien im Base64-Format kodieren bzw. aus dem Base64-Format dekodieren. Recherchieren Sie dazu wie das Format Base64 funktioniert. Sie können die Korrektheit Ihrer Implementierung gegen das Linux-Programm `base64` testen.

Aufgabe 3.9. (Euler-Touren – sehr schwer) Schreiben Sie ein Programm, welches eine Euler-Tour durch einen ungerichteten Graphen findet. Lesen Sie den Graphen von der Standardeingabe als Liste von Zahlenpaaren ein. Jedes Zahlenpaar sagt Ihnen, dass es zwsichen den Knoten mit diesen Zahlen eine Kante gibt. Wie oben beenden Sie die Eingabe mit `-1`. Geben Sie die Tour als Folge von Knoten, die Sie besuchen aus. Gibt es keine Euler-Tour, so geben Sie stattdessen `-1` aus.

Recherchieren Sie mögliche Algorithmen zur Lösung des Problems und wählen Sie eine geeignete Datenstruktur. Es ist ggf. sinnvoll, den Graphen als Array von Listen adjazenter Knoten zu repräsentieren.