Klausur „Grundlagen der Informatik 1 und 2“ Vordiplom Teil A, ET & IT

9. Juli 2002

Hinweise:

- Bitte auf dem Deckblatt an der vorgesehenen Stelle unterschreiben.
- Bitte Namen und Matrikelnummer auf diesem Blatt und auf jedem weiteren Blatt in die Kopfzeile eintragen.
- Bitte keinen Rotstift oder Bleistift verwenden.
- Bitte keine Korrekturmittel verwenden. (Durchstreichen reicht aus.)
- Erreichbare Punktzahl: 100

Aufgabe 1: Programmierung von Listen (20 Punkte)

Im folgenden ist der Code für eine Listenverwaltung mit Hilfe einer einfach verketteten Liste abgedruckt. Er ist angelehnt an den Code für eine doppelt verkettete Liste aus der Vorlesung. Wie dort verwenden wir am Anfang und am Ende der Liste je ein Dummy-Element ohne Daten, um uns die Sonderbehandlung von Anfang und Ende zu ersparen. Die Listenverwaltung bietet bereits die Erzeugung einer Liste mit `createList()` und das Einfügen an den Anfang der Liste mit **`insertToList()`**. Die Funktion `appendToList()` ist noch leer und soll ausgefüllt werden. Sie soll einen Eintrag an das Ende der Liste anhängen, ihn also vor das Dummy-Element am Ende der Liste einfügen.

Dabei soll *nicht* die ganze Liste von vorne an nach der passenden Stelle durchsucht werden. Ändert stattdessen den Aufbau der Liste so, daß der Nachfolger-Zeiger des Dummy-Eintrags am Listenende nicht mehr auf `NULL` zeigt, sondern auf dessen Vorgänger, also auf das letzte benutzte Element (bzw. auf den Dummy-Eintrag am Listenanfang, falls die Liste leer ist). Ändert dazu falls notwendig auch den Code in den vorgegebenen Funktionen durch Hineinschreiben. Hinweis: Bedenkt hierbei insbesondere einen wichtigen Sonderfall!

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {                          /* Listeninhalt: */
    char name[50];                        /* Name */
    unsigned int salary;                  /* Gehalt */
} my_data_t;


struct list_entry_t {                     /* Listeneintrag */
    my_data_t            data;
    struct list_entry_t *next;
};

typedef struct {                          /* Liste */
    struct list_entry_t *headMarker;
    struct list_entry_t *lastMarker;
    struct list_entry_t *currentEntry;
} list_handle_t;
```

```
list_handle_t *createList(void) {
    list_handle_t *handle
            = (list_handle_t *) calloc(1, sizeof(list_handle_t));
    handle->headMarker
       = (struct list_entry_t *) calloc(1, sizeof(struct list_entry_t));
    handle->lastMarker
       = (struct list_entry_t *) calloc(1, sizeof(struct list_entry_t));
    handle->headMarker->next = handle->lastMarker;
    handle->lastMarker->next = NULL;
    handle->currentEntry = handle->headMarker;
    return handle;
}



void insertToList(list_handle_t *handle, my_data_t *d) {
    struct list_entry_t *entry;
    entry = (struct list_entry_t *) calloc(1,sizeof(struct list_entry_t));
    memcpy(&(entry->data), d, sizeof(my_data_t));
    entry->next = handle->headMarker->next;
    handle->headMarker->next = entry;
}



void appendToList(list_handle_t *handle, my_data_t *d) {








}
```

Aufgabe 2: Zahlendarstellung (20 Punkte)

2.1 Abfrage der internen Datenrepräsentation (3 Punkte)

Schreibt die Deklaration eines Datentyps in C, der es erlaubt, seinen Speicherbereich wahlweise entweder als eine `char`-Variable anzusprechen oder als ein Feld von acht einzelnen Bits, also als acht Variablen vom Typ `unsigned` mit der Länge von je einem Bit. Es darf kein Cast benutzt werden, nur `union`, `struct` und Bitfelder. Es soll nur der Datentyp aufgeschrieben werden, keine Funktionen oder Anweisungen.

2.2 Definitionen der Abstraktionsfunktion $\widetilde{\rho}$ und der Addition $\widetilde{+}$ (4 Punkte)

In der Vorlesung wurde das folgende kommutative Diagramm für die Darstellung ganzer Zahlen erläutert:

$$\begin{array}{ccc}
\mathbb{Z} \times \mathbb{Z} & \xrightarrow{+} & \mathbb{Z} \\
\uparrow{\scriptstyle \widetilde{\rho}\times\widetilde{\rho}} & & \uparrow{\scriptstyle \widetilde{\rho}} \\
\mathbb{B}^{n+1} \times \mathbb{B}^{n+1} & \xrightarrow{\widetilde{+}} & \mathbb{B}^{n+1}
\end{array}$$

Schreibt die Definitionen der Abstraktionsfunktion $\widetilde{\rho}$ und der Addition $\widetilde{+}$ auf.

2.3 Beweis einer Eigenschaft des Zweierkomplements (10 Punkte)

Beweist den folgenden Satz:

$$\forall x \in \mathbb{B}^{n+1} \,.\quad K_2(K_2(x)) = x$$

Verwendet dabei die folgende, bekannte Definition:

Definition 1 (Zweierkomplement K_2)

$$K_2 : \mathbb{B}^{n+1} \to \mathbb{B}^{n+1},$$

$$K_2(x_n \;\ldots\; x_0) = \begin{cases} \Pi_{n+1}\big(K_1(x_n \;\ldots\; x_0) +' (0\;0\;\ldots\;0\;1)\big) & \textit{für } (x_n \;\ldots\; x_0) \neq (0\;\ldots\;0) \\ (0\;0\;\ldots\;0) & \textit{für } (x_n \;\ldots\; x_0) = (0\;\ldots\;0) \end{cases}$$

Dabei ist $+'$ die bekannte Addition $+'_{n+1}$ vorzeichenloser Bitvektoren, jetzt aber von $\mathbb{B}^{n+1} \times \mathbb{B}^{n+1}$ nach $\mathbb{B}^{n+2}$ statt von $\mathbb{B}^n \times \mathbb{B}^n$ nach $\mathbb{B}^{n+1}$. Deshalb schneiden wir mit Hilfe der Projektion Π_{n+1} die $(n+2)$-te Stelle ab, um einen Ergebnisvektor aus $\mathbb{B}^{n+1}$ zu bekommen. Dieses dürfen wir tun, weil bei obiger Verwendung gilt, daß diese Ziffer stets gleich 0 ist. □

Weitere Hilfen

Unterscheidet zwei Fälle:

Fall 1: $x = (0\;\ldots\;0)$

Fall 2: $x = (x_n\;\ldots\;x_k\;1\;0\;\ldots\;0)$

2.4 Vergleich Zweierkomplement und Einerkomplement (3 Punkte)

Nennt drei Gründe, warum man heute lieber das Zweierkomplement zur Repräsentation vorzeichenbehafteter Zahlen verwendet als das Einerkomplement:

1.

2.

3.

Aufgabe 3: Programmieren der binären Suche (20 Punkte)

Es ist folgende Header-Datei `binsearch.h` vorgegeben:

```
#include <stdio.h>
typedef struct {
    void *key; /* Zeiger auf Schluessel */
    void *data; /* Zeiger auf Nutzdaten */
} my_array_t;
extern void *binSearch(void           *theKey,
                       my_array_t     *a,
                       unsigned int   len );
extern int compareTo(void *k1, void *k2);
```

Programmiert dazu die Bibliotheksfunktion

```
void *binSearch(void *theKey, my_array_t *a, unsigned int len)
```

Der Anwender dieser Funktion muß einen Zeiger auf ein Array mit sortierten Daten vom Typ `my_array_t` im Parameter `a` übergeben, die Länge dieses Arrays im Parameter `len` und natürlich einen Zeiger auf den zu suchenden Schlüssel im Parameter `theKey`. Die Funktion soll eine binäre Suche auf dem Array nach dem Schlüssel durchführen. Rückgabewert soll ein Zeiger auf die gefundenen Nutzdaten `data` sein, oder `NULL`, falls nichts gefunden wurde. Der Anwender stellt außerdem die Vergleichsfunktion `int compareTo(void *k1, void *k2)` zur Verfügung, die von der Suchfunktion benutzt werden soll. Die Vergleichsfunktion liefert einen beliebigen Wert kleiner als 0 zurück, wenn `*k1` in dieser Vergleichsrelation kleiner als `*k2` ist, einen beliebigen Wert größer als 0, falls `*k1 > *k2`, und bei Gleichheit beider Schlüssel den Wert 0.

Aufgabe 4: BNF/Syntaxdiagramme (10 Punkte)

4.1 EBNF-Grammatik (5 Punkte)

Es sei folgende EBNF-Grammatik vorgegeben:

```
DRINK ::= COCOMINT | COCOLOCO

COCOMINT ::= eis minz kokos asaft osaft { osaft } [ kirsche ]

COCOLOCO ::= eis kokos sahne asaft osaft { msaft } [ raspel ]

eis ::= '8 Eiswürfel '

minz ::= '2cl Pfefferminzsirup '

kokos ::= '2cl Kokossirup '

sahne ::= '2cl Sahne '

osaft ::= 'Orangensaft '

asaft ::= 'Ananassaft '

msaft ::= 'Maracujanektar '

kirsche ::= '1 Cocktailkirsche '

raspel ::= 'Kokosraspel '
```

Das Startsymbol ist `DRINK`.
Schreibt drei von dieser Grammatik akzeptierte Sätze auf.

4.2 Syntaxdiagramm (5 Punkte)

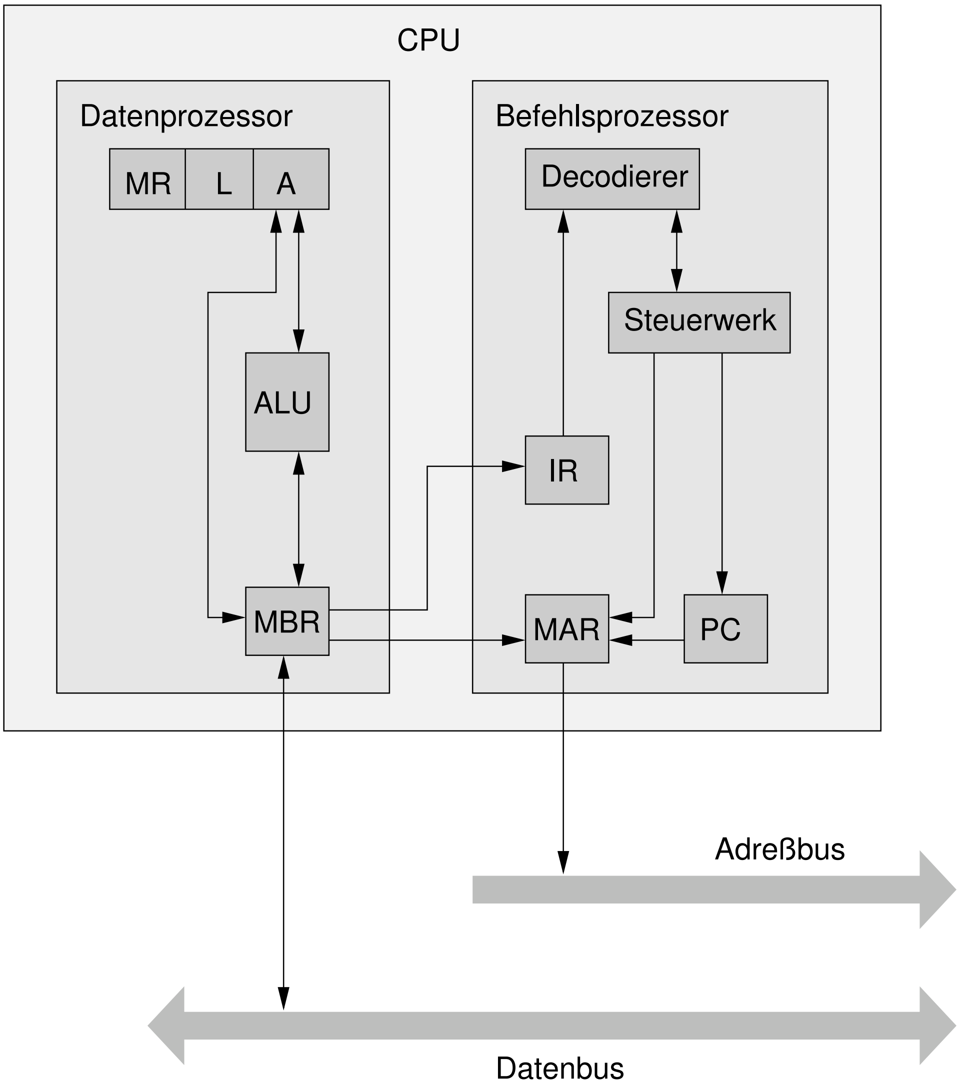
Die folgenden „Sätze“ sind vorgegeben:

- Alkohol
- Allohol
- Alkool
- Alk
- Allool

Zeichnet ein Syntaxdiagramm, das genau diese Sätze aktzeptiert, nicht mehr und nicht weniger. (Es ist nicht zulässig, lediglich fünf Zweige zu zeichnen, die je einen der Sätze akzeptieren.)

Aufgabe 5: von-Neumann-Architektur (10 Punkte)

Das folgende Bild zeigt den Aufbau einer einfachen CPU in der von-Neumann-Architektur:



Beischreibt die Funktion der gezeigten zehn Komponenten der CPU. Soweit Abkürzungen verwendet worden sind, gebt auch eine Langform dafür an.

Aufgabe 6: Verbindungsorientierte Client-Server-Kommunikation (10 Punkte)

Wir wollen verbindungsorientierte Client-Server-Kommunikation mit Sockets in C durchführen. Beschreibt in Textform, wie eine solche Socket-Verbindung aufgebaut und zur Datenübertragung genutzt werden kann, das heißt, welche Partner dafür welche Systemfunktionen aufrufen müssen. Es ist nicht notwendig, die genauen Parameter dieser Funktionen anzugeben, aber die Funktionen müssen aufgezählt und ihre Bedeutung beschrieben werden. Die Erzeugung weiterer Server-Prozesse soll nicht beschrieben werden.

Aufgabe 7: Pseudo-Assembler (10 Punkte)

Setzt das folgende C-Programm-Fragment

```
x=7; y=3; if ( x <= (y+2) ) { z=y; } else { z=0; }
```

mit Hilfe der folgenden Assembler-Befehle um:

`LOAD x`	Lade den Inhalt der Speicheradresse `x` in den Akku
`ADD x`	Addiere den Inhalt der Speicheradresse `x` zum Inhalt des Akkus
`SUB x`	Subtrahiere den Inhalt der Speicheradresse `x` vom Inhalt des Akkus
`JMPNEG x`	Springe zur Speicheradresse `x` falls Akku < 0
`JMP x`	Springe immer zur Speicheradresse `x`
`STORE x`	Speichere den Inhalt des Akkus an die Speicheradresse `x`

Gebt eine entsprechende konkrete Speicherbelegung für den resultierenden Code an, also eine Tabelle aus Speicheradressen und Speicherinhalten. Die Programmausführung startet an der Adresse 0. Nehmt an, daß sowohl ein Befehl als auch eine Adresse als auch eine Zahl je ein Speicherwort belegen.