Lehrstuhl für Informatik III
Prof. Dr.-Ing. M. Nagl

Probeklausur
Grundgebiete der Informatik II

Datum: 16. 06. 2006

Teil II: Algorithmen und Programmiertechniken

Name: *(in Druckschrift)*	
Matr. Nr.:	
Unterschrift	

Aufgabe	Max. Punkte	Korrektur		Einsichtnahme	
		Punkte	Kürzel	Punkte	Kürzel
II.1	10				
II.2	10				
II.3	20				
II.4	20				
Σ	60				

Aufgabe II.1 Wissensfragen und EBNF *(10 Punkte)*

(a) Beschreiben Sie den Unterschied zwischen Call-by-Value und Call-by-Reference. *(1 Punkt)*

(b) Worin besteht der wesentliche Unterschied zwischen einem Feld und einem Verbund? *(1 Punkt)*

(c) Wie ist die Lebensdauer einer lokalen Variablen definiert? *(1 Punkt)*

- ☐ Nur innerhalb des Blocks, in dem die Deklaration steht.
- ☐ Innerhalb und außerhalb des Blocks, in dem die Deklaration steht.
- ☐ Nur außerhalb des Blocks, in dem die Deklaration steht.

(d) Im folgenden soll die EBNF für die einfache Programmiersprache EASY angegeben werden. EASY-Programme sind Programme bestehend aus Anweisungen, in denen einem Bezeichner ein Wert zugewiesen wird (einfache Wertzuweisung). Ein Programm ist folgendermaßen strukturiert:

- Jedes EASY-Programm (program) wird mit dem Schlüsselwort PROGRAM eingeleitet. Anschließend folgt ein Bezeichner (identifier) als Programmname. Nach dem Bezeichner können beliebig viele Anweisungen (assign) vorkommen der oben beschriebenen Form. Alle Anweisungen sind innerhalb eines Paares runder Klammern eingeschlossen.
- Jedes assign besteht aus einem identifier gefolgt von einem Gleichheitszeichen (=). Nach dem Gleichheitszeichen folgt die eigentliche Berechnung, die sich aus zwei Zahlen (number) und einem dazwischenliegenden Operator (operator) zusammensetzt. Jedes assign wird von einem Semikolon (;) abgeschlossen.
- Da es sich hier um eine einfache Programmiersprache handelt, sind als operator nur die Grundrechenarten +, -, * und / zugelassen.
- Neben der Programmstruktur muss der Aufbau einer Zahl (number) und eines Bezeichners (identifier) festgelegt werden. Die Nichtterminalsymbole digit für die Ziffern von 0 bis 9 und letter für alle Buchstaben sind bereits vordefiniert.
  - Jede number ist positiv und besteht aus mindestens einem digit. Anschließend kann optional ein Punkt (.) folgen, hinter dem wieder mindestens ein digit steht.

– Ein Bezeichner (identifier) ist eine Folge von letter und digit, wobei jeder Bezeichner mit einem Buchstaben anfangen muss.

Folgendes Beispiel stellt ein EASY-Programm mit zwei Anweisungen dar.

```
PROGRAM beispiel (
    summe = 16.06 + 2006 ;
    produkt = 5 * 9.2006 ;
)
```

Geben Sie für die Programmiersprache eine geeignete EBNF an. *(7 Punkte)*

Aufgabe II.2 C++-Syntax und -Semantik *(10 Punkte)*

(a) Im folgenden Programm sind insgesamt 3 Syntaxfehler versteckt. Finden Sie diese und ordnen Sie sie in die Kategorien kontextfreie und kontextsensitive Fehler ein. Tragen Sie Ihre Lösung in die nachfolgende Tabelle ein. Geben Sie für jeden Fehler die Zeilennummer, eine kurze Beschreibung des Fehlers und seine Kategorie an. *(6 Punkte)*

```
1  #include <iostream>
2
3  typedef char FeldTyp[100];
4
5  void SortStraightSelection(FeldTyp &feld; int start, int ende) {
6    char elem;
7    int k;
8
9    for (int i = start; i <= ende - 1; i++) {
10     k = i;
11     elem = feld[i];
12     for (int j = i + 1; j <= ende; j++) {
13       if (feld[j] == elem) {
14         k = j;
15         elem = feld[j];
16       }
17     }
18     feld[k] = feld[i];
19     feld[i] = elem;
20   }
21   return 0;
22 }
23
24 int main()  {
25   FeldTyp feld ={ 'e', 't', 'e', 'c', 'h', 'n', 'i', 'k' };
26   SortStraightSelection(feld, 0, 7);
27   for (i = 0; i < 8; i++) {
28     std::cout << feld[i];
29   }
30   return 0;
31 }
```

Zeile	Beschreibung	ks/kf

(b) In folgendem Programm werden die Variablen `y` und `z` als Aktualparameter an die Prozedur `f` übergeben. Geben Sie den Wert von `y` und `z` nach der jeweiligen Ausführung der Prozedur `f` an. *(4 Punkte)*

```
#include <iostream>

void f(int *b, int &c) {
   int a = 2;
   *b += 3;
   c = (*b)*a;
}

int main() {
   int y = 2, z = 4;

   f(&y, z);
  std::cout << "Nach 1. Aufruf: " << " y = " << y << " z = " << z << std::endl;
   f(&y, z);
  std::cout << "Nach 2. Aufruf: " << " y = " << y << " z = " << z << std::endl;

   return 0;
}
```

y	z	
		Wert nach 1. Aufruf
		Wert nach 2. Aufruf

Aufgabe II.3 Bucketsort *(20 Punkte)*

In dieser Aufgabe soll eine Variante des Sortieralgorithmus *Bucketsort* implementiert werden, die ein Feld von Schülern nach Noten sortiert.

(a) Der Verbundtyp `Schueler` besteht aus zwei Komponenten. Die erste Komponente `name` beinhaltet eine Zeichenkette vom Typ `string`. Die zweite Komponente `note` beinhaltet eine ganze Zahl. Geben Sie die Typdefinition des Verbundtyps `Schueler` an. *(2 Punkte)*

(b) Deklarieren Sie eine konstante ganze Zahl `KLASSENGROESSE` und weisen sie dieser den Wert `10` zu. *(1 Punkt)*

(c) Definieren Sie einen Typ namens `KlausurErg_FT`. Dieser ist ein Feld der Größe `KLASSENGROESSE + 1`, dessen Elemente vom Typ `Schueler` sind. *(2 Punkte)*

(d) In dieser Teilaufgabe soll eine Variante des *Bucketsort*-Algorithmus implementiert werden. Der Algorithmus zerfällt in vier Schritte, die als Kommentar im Quelltext beschrieben sind. Setzen Sie die Kommentare in Quelltext um. Sie können dabei die Abbildung, in der beispielhafte Ergebnisse für die Schritte 2 bis 4 abgebildet sind, als gedankliche Hilfe benutzen. *(15 Punkte)*

	0	1	2	3	4	5	6	7	8	9	10
feld	?	Meier 2	Schmid 1	Becker 2	Xu 2	Zimmer 4	Heller 5	Haase 5	Ranger 5	Fuss 6	Held 2

	0	1	2	3	4	5	6
anz (nach 2.)	?	1	4	0	1	3	1
anz (nach 3.)	?	1	5	5	6	9	10

	0	1	2	3	4	5	6	7	8	9	10
sortFeld (nach 4.)	?	Schmid 1	Held 2	Xu 2	Becker 2	Meier 2	Zimmer 4	Ranger 5	Haase 5	Heller 5	Fuss 6

```
void Bucketsort(KlausurErg_FT & feld, KlausurErg_FT & sortFeld,
                int feldgroesse) {
// HINWEIS: Alle Felder werden im Folgenden aus Einfachheits-
// gründen AB INDEX 1 belegt und durchlaufen

// 1) Deklaration eines Ganzzahlfeldes 'anz' der Größe 7 und
//    Initialisierung mit 0 für jedes Element (außer 'anz[0]')
```

```
// 2) Berechnung der Notenhäufigkeiten in 'feld'. 'feld' einmal
//    durchlaufen und dabei für jede gefundene Note 'j' das
//    Element 'anz[j]' um eins erhöhen.
```

```
    // 3) Anzahlen aufsummieren, so dass anschließend in 'anz[j]' die
    // Zahl der Schueler mit Note j ODER BESSER (also kleinergleich j)
    // steht.









    // 4) Personen von 'feld' mittels 'anz' an richtige Stelle in
    // 'sortFeld' kopieren. Die richtige Stelle für eine Note j
    // steht dabei nun in 'anz[j]'. Nach dem Kopieren muss
    // 'anz[j]' um eins erniedrigt werden.









}
```

Aufgabe II.4 ADO Adressliste *(20 Punkte)*

In dieser Aufgabe soll ein abstraktes Datenobjektmodul (ADO **Adressliste**) zum Speichern von Adressen implementiert werden. Dabei handelt es sich um eine sortierte Liste, die Datensätze, bestehend aus Name, Straße und Stadt, enthält. Der Name ist das Sortierkriterium. Das ADO soll die folgende Schnittstelle anbieten:

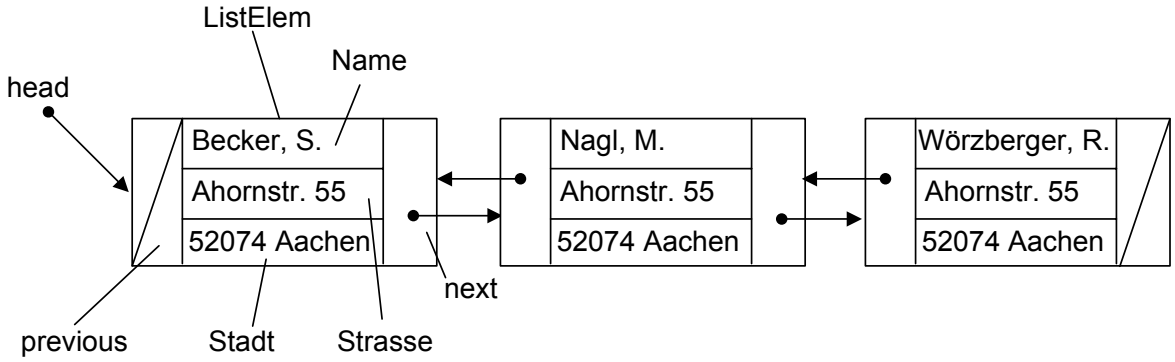
```
// Adressen.h
// ADO Adressliste

// Initialisiert die Datenstruktur.
void Addrs_init();

// Fuegt einen Datensatz ein.
void Addrs_insert(string name, string strasse, string stadt);

// Gibt die Liste auf der Standardausgabe aus.
void Addrs_dump();
```

Die folgende Abbildung verdeutlicht den internen Aufbau der Datenstruktur an einem Beispiel:



Die Liste ist mit Hilfe von verzeigerten Elementen vom Typ **ListElem** auf der Halde realisiert. Der Zeiger **head** zeigt auf das erste Element. Jedes Element besteht aus einem Eintrag für den Namen (**Name**), für die Straße (**Strasse**) und für die Stadt (**Stadt**). Alle drei sind vom Typ **string**. Die Elemente sind untereinander über den Zeiger **next** vorwärts und über den Zeiger **previous** rückwärts verkettet. Es gibt kein leeres Element am Anfang der Liste, um Spezialfälle beim Einfügen oder Löschen von Elementen zu vermeiden. Das erste Element enthält, wenn vorhanden, stets Daten; die leere Liste enthält kein Element. Im letzten Element der Liste enthält der Zeiger **next** den Wert NULL, im ersten Element der Liste enthält der Zeiger **previous** den Wert NULL.

(a) Vervollständigen Sie die Typdeklaration von **ListElem**. Dann deklarieren Sie den Kopfzeiger. Verwenden Sie die Bezeichner aus der Abbildung! *(3 Punkte)*

```
struct ListElem {




};
```

(b) Vervollständigen Sie die Prozedur **Addrs_insert** (s. nächste Seite), die einen neuen Datensatz an der richtigen Stelle in eine bereits sortierte Liste einfügt, so dass diese danach immer noch sortiert ist. Die Implementierung verwendet die folgende lokale Hilfsfunktion:

```
// Gibt das Listenelement zurück, nach dem
// ein neues Element mit dem Namen
// 'name' einzufügen ist, d.h. das Vorgänger-
// Listenelement. Ist das neue Element
// am Anfang der Liste einzufügen, ist der
// Rückgabewert NULL.
ListElem *findElementBefore(string name);
```

Gehen Sie davon aus, dass noch kein Element mit dem selben Namen existiert. Beachten Sie die doppelte Verkettung. Diese führt u.a. zu Sonderfällen beim Einfügen des ersten und des letzten Elements. *(10 Punkte)*

```
void Addrs_insert(string name, string strasse, string stadt) {
   ListElem *newElem;

   // neues Element erzeugen und initialisieren
   newElem=new ListElem;

   newElem->Name=name;
   newElem->Strasse=strasse;
   newElem->Stadt=stadt;

   // Einfuegestelle suchen
   ListElem *before=findElementBefore(name);

   if (before!=NULL) {
      // Einfügen hinter dem Listenelement 'before'









   } else {
      // Einfügen als erstes Element









   }
}
```

(c) Ergänzen Sie nun die Implementierung der Funktion **findElementBefore**. Auch hier können Sie davon ausgehen, dass es noch kein Element mit dem übergebenen Namen gibt. Für den lexikographischen Vergleich von Zeichenketten vom Typ **string** können Sie den Operator < verwenden. *(7 Punkte)*

```
ListElem *findElementBefore(string name) {




}
```