Fachhochschule Wedel
Prof. Dr. Uwe Schmidt

5. Februar 1998

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im WS 97/98 (WI63)

Zeit: 150 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Zwischen der Bearbeitung der Aufgaben 3, 4 und 5 über Java und der Aufgaben 6 und 7 über C++ kann gewählt werden. Wird eine der Aufgaben 3, 4 oder 5 bearbeitet, so werden nur diese gewertet, sonst werden Aufgaben 6 und 7 gewertet.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 21 Seiten

---

Aufgabe 1:

Gegeben sei das folgende C–Programmstück für die Verarbeitung von binären Bäumen:

```
#include <stdlib.h>
#include <string.h>

typedef char * String;

typedef struct node * Bintree;
struct node {
    String info;
    Bintree l;
    Bintree r;
};

#define empty ((Bintree)0)

Bintree mk1(String e) {
    Bintree t;
    t = malloc(sizeof(*t));
    if ( !t )
        exit(1);
    t->info = e;
    t->l = empty;
    t->r = empty;
    return t;
}

int compare(String e1, String e2) {
    int c = strcmp(e1,e2);
    return
        ( c > 0) ? 1 : (c == 0) ? 0 : -1;
}

extern Bintree insert(String e1, Bintree t);
```

Implementieren Sie die fehlende *insert* Routine. Diese trägt ein Element, hier eine Zeichenreihe, in einen binären Baum ein, falls das Element noch nicht in dem Baum vorhanden ist.

Nutzen Sie die in dem Programmstück vorgegebenen Datentypen, Makros und Funktionen.

(Lösung auf der nächsten Seite).

Die *insert* Routine:

Aufgabe 2:

Gegeben sei das folgende Programm:

```
#include <stdio.h>

typedef struct entry *dll;
struct entry {
    int i;
    dll n;
    dll p;
};

struct entry e[3] = {
                        {-3, e + 1, &e[2]},
                        {0, e + 2, &e[0]},
                        {5, e, &e[1] }
};

dll l = e;
dll *pl = &l;
int cnt = 0;

int main (int argc, char *argv[])
{
  printf ("%d) %d\n", ++cnt, l→i);
  printf ("%d) %d\n", ++cnt, l→p→i);
  printf ("%d) %d\n", ++cnt, l→p→p→i);
  printf ("%d) %d\n", ++cnt, l→p→p→p→i);
  printf ("%d) %d\n", ++cnt, l→n→n→n→i);
  printf ("%d) %d\n", ++cnt, l→n→p→p→p→n→n→i);

  printf ("%d) %d\n", ++cnt, (l++)→n→i);
  printf ("%d) %d\n", ++cnt, (l++)→p→i);
  l = e + 2;
  printf ("%d) %d\n", ++cnt, (--l)→p→i);
  printf ("%d) %d\n", ++cnt, (--l)→n→i);
  l = e + 1;
  printf ("%d) %d\n", ++cnt, l[-1].n→i);
  printf ("%d) %d\n", ++cnt, l[0].i);
  printf ("%d) %d\n", ++cnt, l[+1].p→i);
  l = e;
  printf ("%d) %d\n", ++cnt, (**pl).i);

  return 0;
}
```

Welche Ausgabezeilen liefert dieses Programm:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Aufgabe 3:

Gegeben sei die folgende Java-Klasse:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public
class ButtonEvent extends Applet {

  Button b;

  public
  void init () {
    b = new Button("drücke mich");
    add(b);

    // event-Behandlungsroutine
    // wird hier eingesetzt
  }
}
```

Erweitern Sie dies Klasse so, daß sie eine event-Behandlungsroutine für einen Knopfdruck auf den button *b* implementiert. Der handler soll auf die Standardausgabe die Zeichenreihe *"autsch"* ausgeben. *ActionListener* ist die zugehörige event-listener Klasse für *ActionEvents*, *actionPerformed* der Name der gerufenen Methode.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 4:

Verarbeitung von arithmetischen Ausdrücken mit **Java**:

Es soll eine Klassenhierarchie zur Verarbeitung von arithmetischen Ausdrücken entwickelt werden. Der Einfachheit halber sollen hier nur Ausdrücke über den ganzen Zahlen betrachtet werden.

Die Wurzelklasse diese Klassenhierarchie sei die Klasse *Expr*:

```
//—————————

public
abstract
class Expr {

  // die eval Funktion

  public
  abstract
  int eval();

}

//—————————
```

Die einzige Verarbeitungsmethode ist hier das Auswerten eines Ausdrucks, die *eval*–Funktion.

Arithmetische Ausdrücke sind im einfachsten Fall Konstante, in dieser Aufgabe also ganze Zahlen. Für Konstante wird eine Klasse *Const* benötigt. Geben sie eine sinnvolle Definition für die *Const*–Klasse an:

Zusammengesetzte Ausdrücke sind z.B. einstellige und zweistellige Ausdrücke. In dieser Aufgabe sollen nur Ausdrücke der Form $e_1 + e_2$ und $e_1 * e_2$ behandelt werden. Entwickeln Sie sinnvolle Definitionen für allgemein verwendbare Klassen $AddExpr$ für Addition und $MultExpr$ für die Multiplikation. Gemeinsame Teile der Klassendefinition sind gegebenenfalls zusammenzufassen.

(*AddExpr* und *MultExpr* cont.)

Erweitern Sie die Klassenhierarchie so, daß auch Subtraktionen verarbeitet werden können.

Aufgabe 5:

Gegeben sei das folgende Testprogramm:

```
public
class ExcTest {
  public
  static
  void main(String[] argv) {
    int i = 0;
    try {
      i = Integer.parseInt(argv[0]);
    }
    catch ( Exception e ) { }
    test(i);
  }

  public
  static
  void test(int i) {
  tryblock:
    try {
      switch ( i ) {
      case 1:
        System.out.println("test:  call foo");
        foo();
        break;
      case 2:
        System.out.println("test:  call wow");
        wow();
        break;
      default:
        System.out.println("test:  normal exit");
      }
    }
    catch ( MyException e ) {
      System.out.println(
        "test:  caught MyException");
    }
    catch ( Exception e ) {
      System.out.println("test:  caught Exception");
    }
    finally {
      System.out.println("test:  exec finally");
    }
    System.out.println("test:  normal exit");
  }
```

```
  public
  static
  void foo()
    throws MyException {
      try {
        System.out.println(
          "foo :  throw MyException");
        throw
          new MyException("foo test");
      }
      finally {
        System.out.println("foo :  exec finally");
      }
  }

  public static
  void wow()
    throws MyException {
      int a, b=1, c=0;
      try {
        System.out.println(
          "wow :  division by 0");
        a = b/c;
      }
      catch (Exception e) {
        System.out.println(
          "wow :  caught Exception");
        System.out.println(
          "wow :  throw MyException");

        throw
          new MyException("wow test");
      }
  }
}

class MyException extends Exception {
  public
  MyException(String s) {
    super(s);
  }
}
```

Welche Ausgabe wird bei einem Aufruf von `java ExcTest 1` erzeugt:

.....

.....

.....

.....

.....

.....

.....

Welche Ausgabe wird bei einem Aufruf von `java ExcTest 2` erzeugt:

.....

.....

.....

.....

.....

.....

.....

Aufgabe 6:

Verarbeitung von arithmetischen Ausdrücken mit **C++**:

Es soll eine Klassenhierarchie zur Verarbeitung von arithmetischen Ausdrücken entwickelt werden. Der Einfachheit halber sollen hier nur Ausdrücke über den ganzen Zahlen betrachtet werden.

Der eigentliche *Expr*-Datentyp ist ein Zeiger-Datentyp, die zugehörige Klasse ist *ExprNode*. Diese Klasse bildet die Wurzelklasse einer Klassenhierarchie für unterschiedliche arithmetische Ausdrücke:

```
//——————————
// der Zeiger-Datentyp für arithmetische Ausdrücke

typedef class ExprNode * Expr;


// die zugehörige Klasse
class ExprNode {

  // die eval Funktion

public:
  virtual
  int eval() = 0;

  virtual
  ~ExprNode() = 0;
};

//——————————
```

Die einzige Verarbeitungsmethode ist hier das Auswerten eines Ausdrucks, die *eval*–Funktion. Zusätzlich ist aber noch ein Destruktor für die Speicherverwaltung notwendig.

Arithmetische Ausdrücke sind im einfachsten Fall Konstante, in dieser Aufgabe also ganze Zahlen. Für Konstante wird eine Klasse $ConstNode$ benötigt. Geben sie eine sinnvolle Definition für die $ConstNode$–Klasse an:

Zusammengesetzte Ausdrücke sind z.B. einstellige und zweistellige Ausdrücke. In dieser Aufgabe sollen nur Ausdrücke der Form $e_1 + e_2$ und $e_1 * e_2$ behandelt werden. Entwickeln Sie sinnvolle Definitionen für allgemein verwendbare Klassen $AddExprNode$ für Addition und $MultExprNode$ für die Multiplikation. Gemeinsame Teile der Klassendefinition sind gegebenenfalls zusammenzufassen.

(*AddExprNode* und *MultExprNode* cont.)

Erweitern Sie die Klassenhierarchie so, daß auch Subtraktionen verarbeitet werden können.

Aufgabe 7:

Gegeben sei das folgende Programm, in dem mehrfache Vererbung verwendet wird:

```
#include <iostream.h>

int xc = 0;

class X {
  int xi;
public:
  X(int i = 42) : xi(i) { cout << " X: " << xi << endl; }
  ~X() { cout << " X: " << xi << endl; }
};

class Y : public X {
  int yi;
public:
  Y(int i) : X(2*i), yi(i) { cout << " Y: " << yi << endl; }
  ~Y() { cout << " Y: " << yi << endl; }
};

class Z : public X {
  int zi;
public:
  Z(int i) : X(5*i), zi(i) { cout << " Z: " << zi << endl; }
  ~Z() { cout << " Z: " << zi << endl; }
};

class P : public Y, public Z {
  int pi;
public:
  P(int i) : Y(3*i), Z(2*i), pi(i) { cout << " P: " << pi << endl; }
  ~P() { cout << " P: " << pi << endl; }
};

int main() {
  {
    P p(1);
    cout << "-----" << endl;
  }

  return 0;
}
```

Welche Ausgabe erzeugt dieses Programm

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
