Name: _____ Matrikelnr.: _____

Hinweise zur Bearbeitung der Klausur zum Kurs 01613 „Einführung in die imperative Programmierung"

1. Prüfen Sie die Vollständigkeit Ihrer Unterlagen. Die Klausur umfasst:
   - 2 Deckblätter
   - 1 Formblatt für eine Bescheinigung für das Finanzamt
   - diese Hinweise zur Bearbeitung
   - 4 Aufgaben (Seite 2-16)
   - die Muss-Regeln des Programmierstils
2. Füllen Sie, **bevor** Sie mit der Bearbeitung der Aufgaben beginnen, folgende Seiten des Klausurexemplares aus:
   - Beide Deckblätter mit Namen, Anschrift sowie Matrikelnummer.
   - Falls Sie eine Teilnahmebescheinigung für das Finanzamt wünschen, füllen Sie bitte das entsprechende Formblatt aus und belassen Sie es in der Klausur. Sie erhalten es dann zusammen mit der Korrektur abgestempelt zurück.

   Nur wenn Sie die Deckblätter vollständig ausgefüllt haben, werden wir Ihre Klausur korrigieren!
3. Schreiben Sie Ihre Lösungen jeweils auf den freien Teil der Seite unterhalb der Aufgabe bzw. auf die leeren Folgeseiten. Sollte dies nicht möglich sein, so vermerken Sie, auf welcher Seite die Lösung zu finden ist.
4. **Streichen Sie ungültige Lösungen deutlich durch!** Sollten Sie mehr als eine Lösung zu einer Aufgabe abgeben, so wird nur eine davon korrigiert – und nicht notwendig die bessere.
5. Schreiben Sie auf jedes von Ihnen beschriebene Blatt oben links Ihren Namen und oben rechts Ihre Matrikelnummer. Wenn Sie weitere eigene Blätter benutzt haben, heften Sie auch diese, mit Namen und Matrikelnummer versehen, an Ihr Klausurexemplar. Nur dann werden auch Lösungen außerhalb Ihres Klausurexemplares gewertet!
6. Neben unbeschriebenem Konzeptpapier und Schreibzeug (Füller oder Kugelschreiber, benutzen Sie **keinen Bleistift** und **keinen Rotstift**!) sind **keine weiteren Hilfsmittel** zugelassen.
7. Es sind maximal 24 Punkte erreichbar. Sie haben die Klausur bestanden, wenn Sie mindestens 12 Punkte erreicht haben.

Aufgabe 1 (6 Punkte)

Gegeben ist das folgende Programm `WasPassiert`.

```
program WasPassiert(input,output);
  var
  a:integer;
  b:integer;

  function check(a,b:integer):boolean;
  begin
    while (b > 0) do
      b := b - a;
    check := (b = 0)
  end;

begin
  readln(a);
  readln(b);
  writeln(check(a,b))
end.
```

Geben Sie eine Problemspezifikation an, die durch die Funktion `check` gelöst wird.

Eingabe:

Ausgabe:

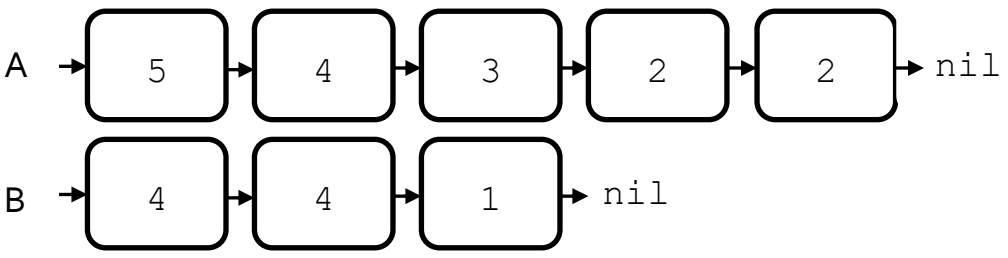
Nachbedingung:

Name: _____ Matrikelnr.: _____

Aufgabe 2 (6 Punkte)

Gegeben sind die folgenden Typendefinitionen, die Funktion `sub` und zwei Listen `A` und `B`.

```
type
tRefListe = ^tListe;
tListe = record
           wert: integer;
           next: tRefListe
         end;

function sub(inListeA: tRefListe; inListeB: tRefListe): tRefListe;
  var
  ListeC: tRefListe;
begin
  if (inListeA = nil) then
    sub := nil
  else
  if (inListeB = nil) then
  begin
    new(ListeC);
    ListeC^.wert := inListeA^.wert;
    ListeC^.next := sub(inListeA^.next, nil);
    sub := ListeC
  end
  else
    if inListeA^.wert - inListeB^.wert < 0 then
      sub := sub(inListeA^.next, inListeB^.next)
    else
    begin
      new(ListeC);
      ListeC^.wert := inListeA^.wert - inListeB^.wert;
      ListeC^.next := sub(inListeA^.next, inListeB^.next);
      sub := ListeC
    end
end;
```



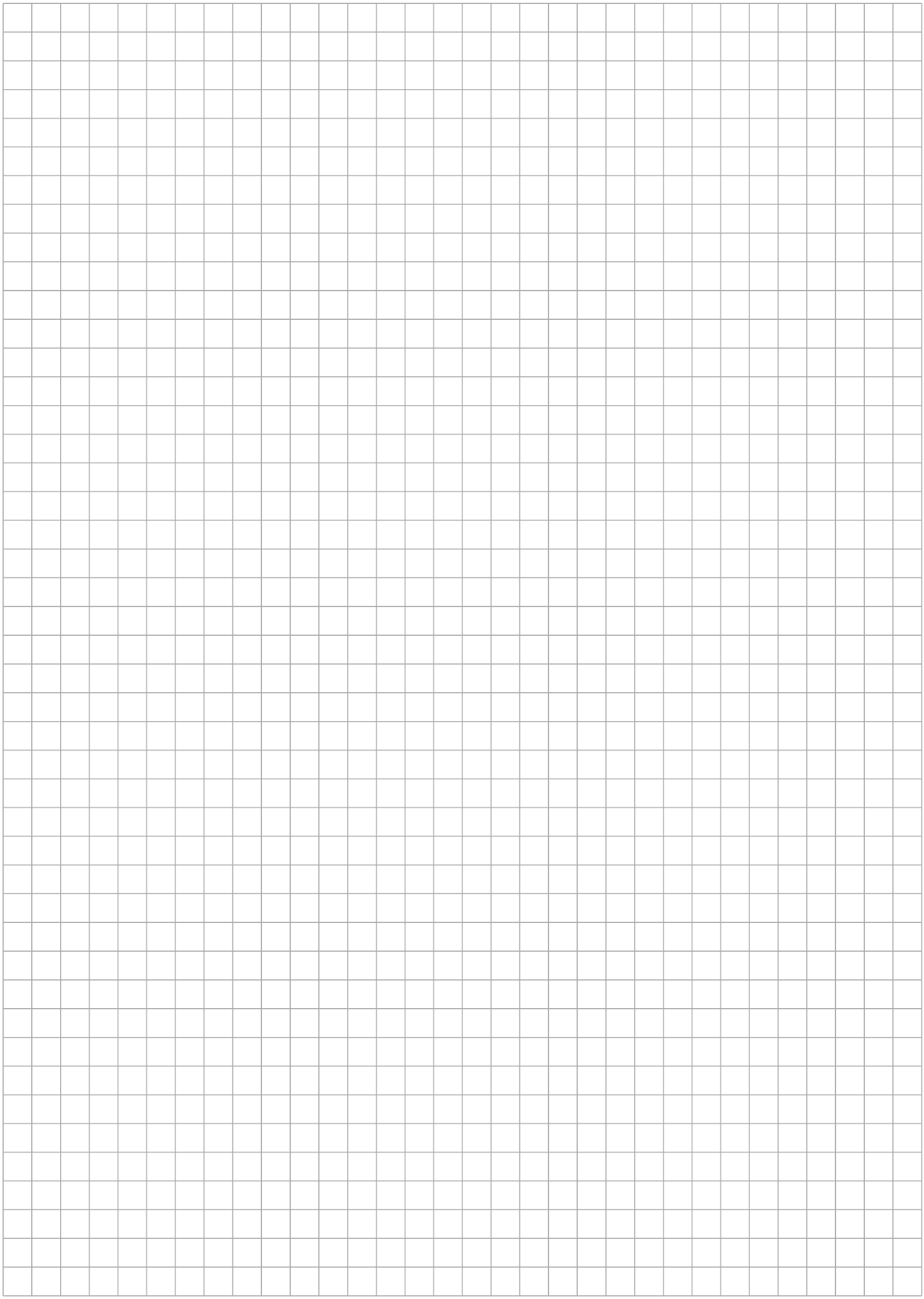
Name: _____ Matrikelnr.: _____

Wie sehen zwei Listen `C` und `D` nach Aufruf der folgenden zwei Befehle aus?

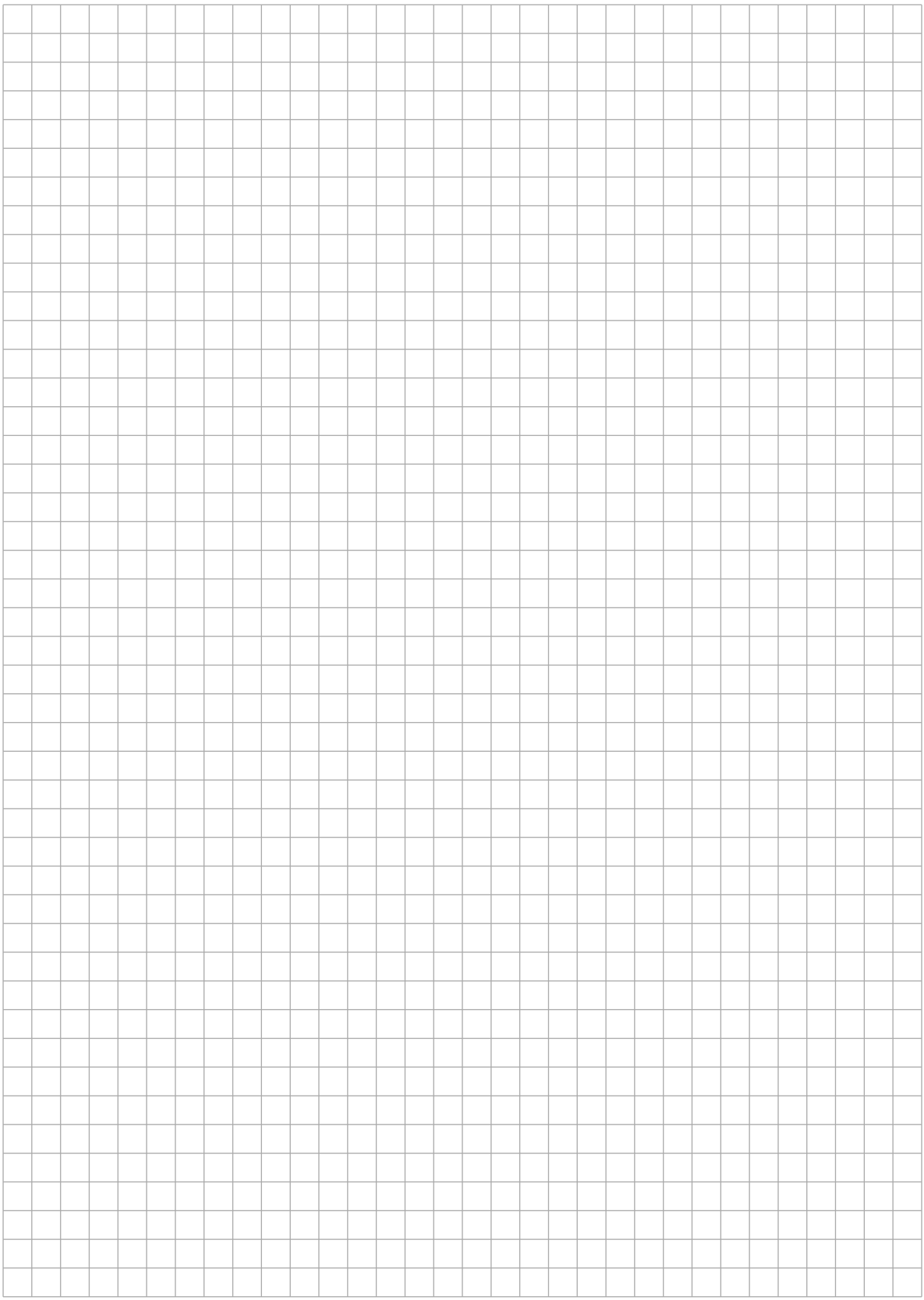
```
C := sub(A,B);
D := sub(C,B);
```

C →

D →



Name: _____ Matrikelnr.: _____



Aufgabe 3 (6 Punkte)

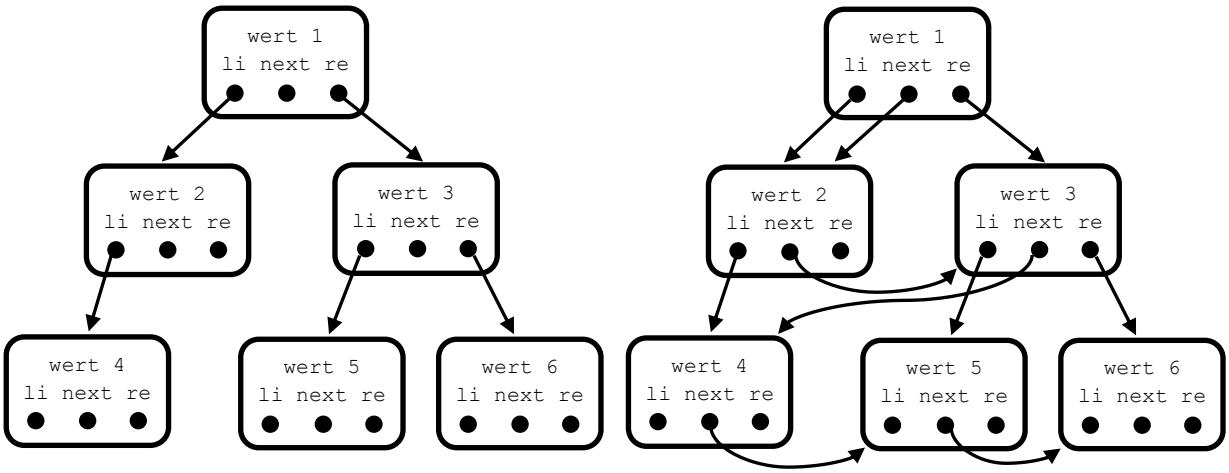
Gegeben sind die folgenden Typendefinitionen für Binärbäume aus integer-Zahlen mit einer zusätzlichen Referenz `next` und die zwei Prozeduren `setLevelOrder` und `levelOrder`.

```
type
tRefBinBaum = ^tBinBaum;
tBinBaum = record
             wert:integer;
             li:tRefBinBaum;
             re:tRefBinBaum;
             next:tRefBinBaum
           end;
```

`setLevelOrder` benutzt die Referenzen `li` und `re` eines Binärbaums, um für den Baum die Zeilenordnung zu berechnen und die Referenzen `next` entsprechend dieser Ordnung zu setzen. Die Zeilenordnung durchläuft einen Binärbaum für jede Tiefe des Baumes von links nach rechts.

`levelOrder` benutzt die gesetzten Referenzen `next`, um die Werte eines Binärbaums in Zeilenordnung auszugeben.

Als Beispiel betrachten wir auf der linken Seite einen Binärbaum, dessen Werte hier zufälligerweise der Zeilenordnung entsprechen. Auf der rechten Seite ist derselbe Binärbaum nach dem Aufruf der Prozedur `setLevelOrder` dargestellt. Nun führt der Aufruf von `levelOrder` zu der Ausgabe `1,2,3,4,5,6`.

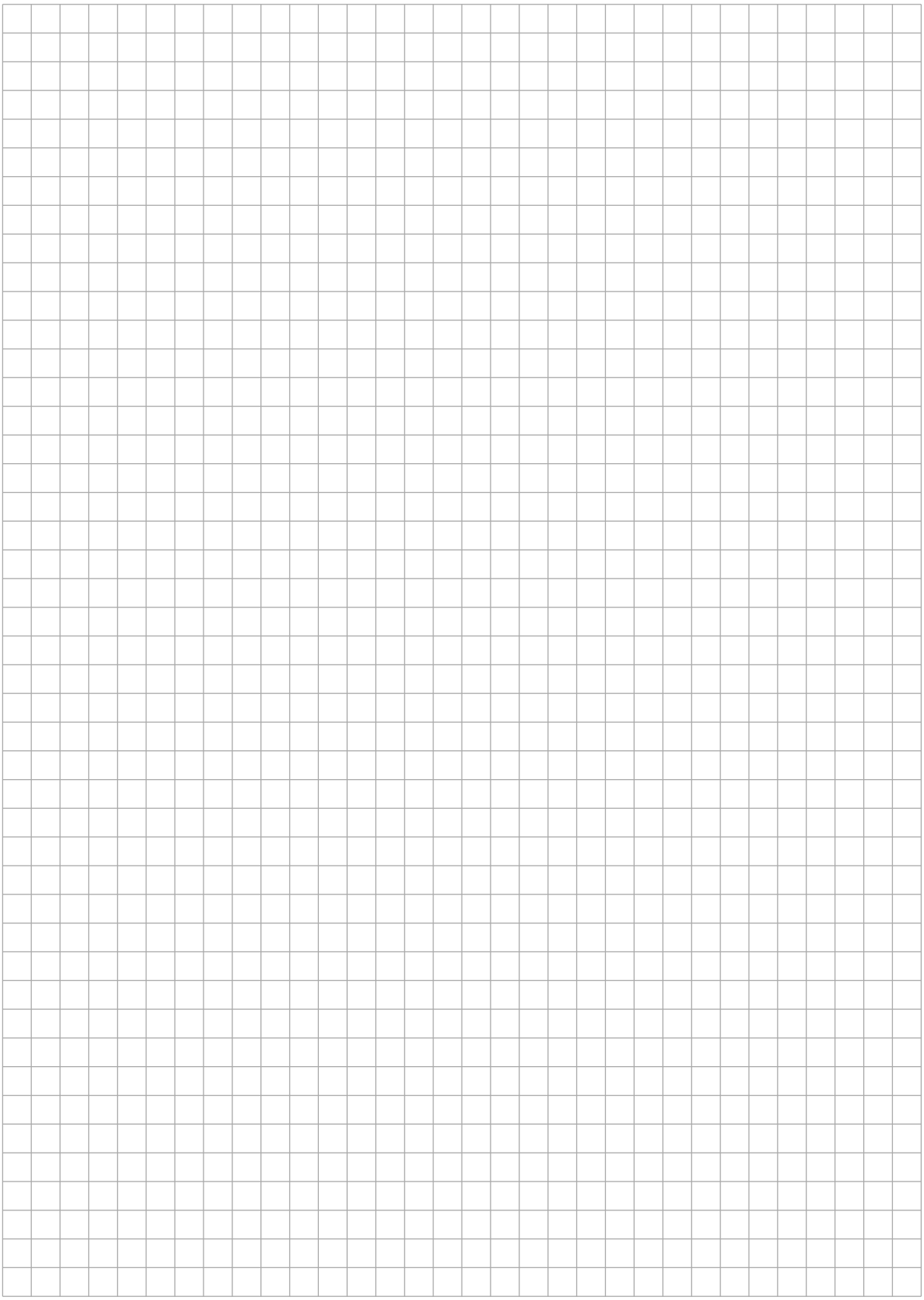


Name: _____ Matrikelnr.: _____

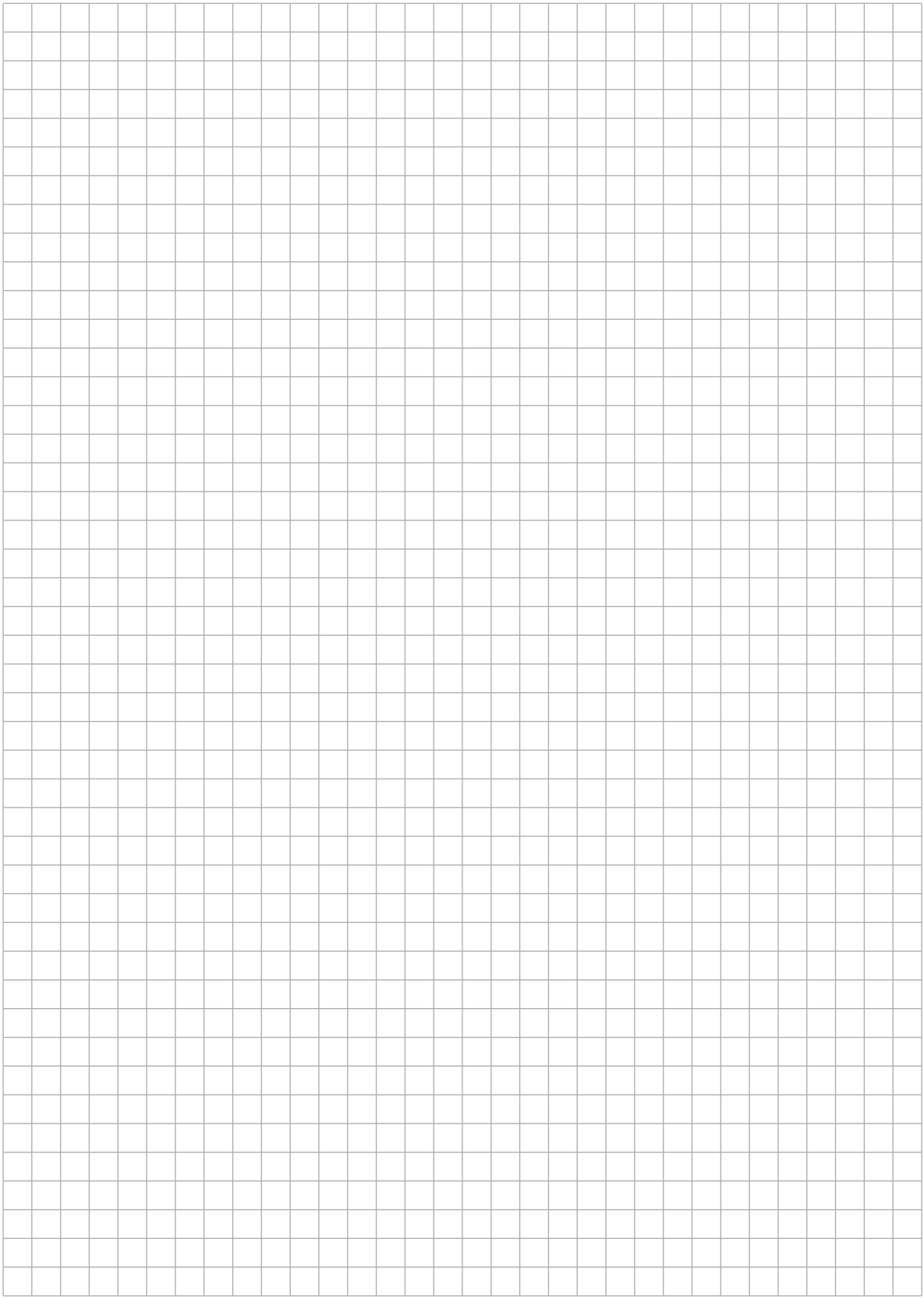
Hier finden Sie die Prozeduren `setLevelOrder` und `levelOrder`. Die Prozeduren sind jedoch noch unvollständig. Ergänzen Sie beide passend, an den grau eingefärbten Stellen, jeweils um eine Zeile.

```
procedure setLevelOrder(inBaum:tRefBinBaum);
  var
  lauf       :tRefBinBaum;
  listenEnde :tRefBinBaum;

begin
  lauf := inBaum;

  while (lauf <> nil) do
  begin
    if (lauf^.li <> nil) then
    begin
      listenEnde^.next := lauf^.li;
      listenEnde := listenEnde^.next;
    end;
    if (lauf^.re <> nil) then
    begin
      listenEnde^.next := lauf^.re;
      listenEnde := listenEnde^.next;
    end;

  end;
end;
```

```
procedure levelOrder(inBaum:tRefBinBaum);
begin
  if (inBaum <> nil) then
  begin
    writeln(inBaum^.wert);

  end
end;
```



Name: _____ Matrikelnr.: _____

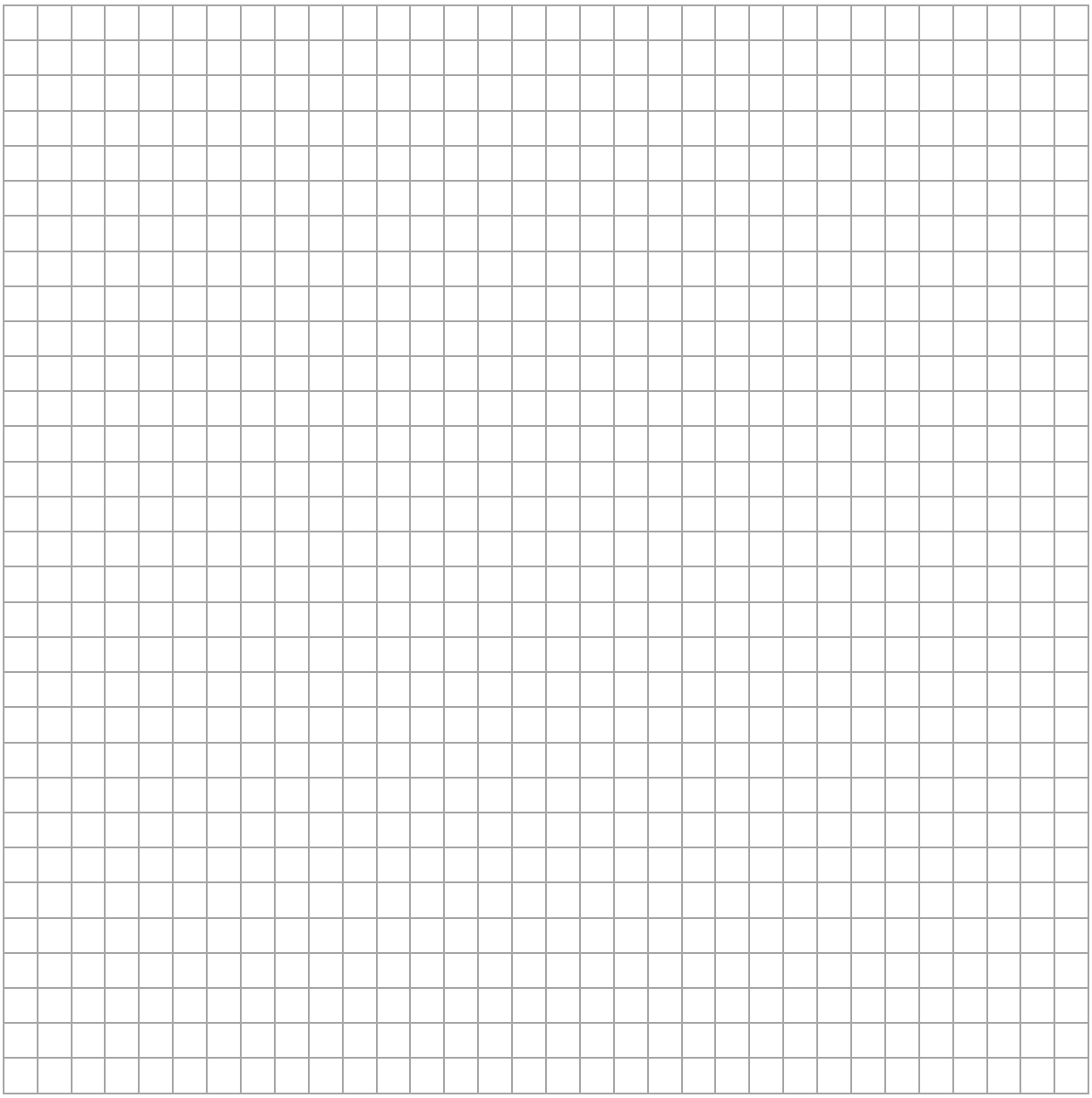


Aufgabe 4 (6 Punkte)

Gegeben sind die folgenden Typendefinitionen und die Funktion `add` auf der nächsten Seite.

Geben Sie drei Testdaten an, die zusammen alle Anweisungen der Funktion `add` überdecken.

```
type
tRefIndexListe = ^tIndexListe;
tIndexListe = record
                index:integer;
                wert:String;
                next:tRefIndexListe
              end;
```



Name: _____ Matrikelnr.: _____

```
procedure add(inIndex:integer; inWert:String;
              var ioListe:tRefIndexListe);
  var
  akt:tRefIndexListe;
  neu:tRefIndexListe;
  gefunden:boolean;
begin
  new(neu);
  neu^.index := inIndex;
  neu^.wert := inWert;
  if (inIndex <= ioListe^.index) then
  begin
    neu^.next := ioListe;
    ioListe := neu;
  end
  else
  begin
    akt := ioListe;
    gefunden := false;
    while ((akt^.next <> nil) and not gefunden) do
    begin
      if ((akt^.next^.index) >= inIndex) then
      begin
        neu^.next := akt^.next;
        akt^.next := neu;
        gefunden := true
      end;
      akt:=akt^.next
    end;
    if (not gefunden) then
    begin
      akt^.next := neu;
      neu^.next := nil
    end
  end;
  akt := ioListe;
  while (akt^.next <> nil) do
  begin
    if (akt^.index = akt^.next^.index) then
      akt^.next^.index := akt^.next^.index + 1;
    akt := akt^.next
  end
end;
```

Name: _____ Matrikelnr.: _____

Kurs 01613 „Einführung in die imperative Programmierung"

Zusammenfassung der Muss-Regeln

1. Selbstdefinierte Konstantenbezeichner bestehen nur aus Großbuchstaben. Bezeichner von Standardkonstanten wie z.B. `maxint` sind also ausgenommen.
2. Typenbezeichnern wird ein `t` vorangestellt. Bezeichnern von Zeigertypen wird ein `tRef` vorangestellt. Bezeichner formaler Parameter beginnen mit `in`, `io` oder `out`.
3. Jede Anweisung beginnt in einer neuen Zeile. `begin` und `end` stehen jeweils in einer eigenen Zeile.
4. Anweisungsfolgen werden zwischen `begin` und `end` um eine konstante Anzahl von 2-4 Stellen eingerückt. `begin` und `end` stehen linksbündig unter der zugehörigen Kontrollanweisung, sie werden nicht weiter eingerückt.
5. Anweisungsteile von Kontrollanweisungen werden genauso eingerückt.
6. Im Programmkopf wird die Aufgabe beschrieben, die das Programm löst.
7. Jeder Funktions- und Prozedurkopf enthält eine knappe Aufgabenbeschreibung als Kommentar. Ggf. werden zusätzlich die Parameter kommentiert.
8. Die Parameter werden sortiert nach der Übergabeart: Eingangs-, Änderungs- und Ausgabeparameter.
9. Die Übergabeart jedes Parameters wird durch Voranstellen von `in`, `io` oder `out` vor den Parameternamen gekennzeichnet.
10. Das Layout von Funktionen und Prozeduren entspricht dem von Programmen.
11. Jede von einer Funktion oder Prozedur benutzte bzw. manipulierte Variable wird als Parameter übergeben. Es werden keine globalen Variablen manipuliert.
12. Jeder nicht von der Prozedur veränderte Parameter wird als Wertparameter übergeben. Lediglich Felder können auch anstatt als Wertparameter als Referenzparameter übergeben werden, um den Speicherplatz für die Kopie und den Kopiervorgang zu sparen. Der Feldbezeichner beginnt aber stets mit dem Präfix `in`, wenn das Feld nicht verändert wird.
13. Pascal-Funktionen werden wie Funktionen im mathematischen Sinne benutzt, d.h. sie besitzen nur Wertparameter. Wie bei Prozeduren ist eine Ausnahme nur bei Feldern erlaubt, um zusätzlichen Speicherplatz und Kopieraufwand zu vermeiden.
14. Wertparameter werden nicht als lokale Variable missbraucht.
15. Die Laufvariable wird innerhalb einer `for`-Anweisung nicht manipuliert.
16. Die Grundsätze der strukturierten Programmierung sind strikt zu befolgen.