Dr. S. Lewandowski **Einführung in die Informatik I (autip)** WS 2006/07

**Übungsblatt 06**

Ausgabe: 06.12. Abgabeschluss: Mittw., 13.12., 9:45 Uhr, `eClaus.informatik.uni-stuttgart.de`

---

Abgabe erfolgt ausschließlich elektronisch über `eClaus.informatik.uni-stuttgart.de` – versuchen Sie nach Möglichkeit die Abgabe nicht in der letzten Minute zu machen!

Von jedem Aufgabenblatt werden maximal 20 Punkte auf den Schein angerechnet.

---

1. (1+1 Punkte) **Trick 17:** In einem Programm eines Kommilitonen sehen Sie folgende Anweisungen für die beiden `integer`-Variablen `a` und `b`: `b:=b-a; a:=a+b; b:=a-b;`

   (1 Punkt) Was bewirken diese drei Zuweisungen? (1 Punkt) Bewerten Sie dieses Vorgehen.

2. (11(+6) Punkte) **Binärzahlen:** Diese Aufgabe soll uns ein tieferes Verständnis der Datentypen im Inneren unseres Rechners vermitteln.

   Im Rechner sind alle Daten durch Vektoren von Bits (mit Wert 0 oder 1) dargestellt. Wir wollen hier nun die Addition von positiven ganzen Zahlen mit 8 Bits simulieren. Mit 8 Bits lassen sich $2^8 = 256$ verschiedene Werte darstellen. Bei der gewöhnlichen Darstellung von Zahlen zur Basis 2 ist der Wert einer Zahl in Binärdarstellung $b_7b_6b_5b_4b_3b_2b_1b_0 = \sum_{i=0}^{7} 2^i \cdot b_i$, so hat 00010110 den Wert $0\cdot2^7+0\cdot2^6+0\cdot2^5+1\cdot2^4+0\cdot2^3+1\cdot2^2+1\cdot2^1+0\cdot2^0 = 16+4+2 = 22$.

   Bei der Addition zweier Zahlen geht man wie beim schriftlichen Addieren zweier Dezimalzahlen stellenweise vor (beginnend bei der niedrigstwertigen Stelle). Addieren wir zu 00010110 eine 1, so erhalten wir 00010111. Addieren wir zu dieser Zahl nochmals 1, so ergibt die Addition auf die letzte Stelle einen Überlauf, der auf die Stelle davor übertragen wird. Dort steht jedoch auch eine 1, sodass erneut ein Überlauf passiert usw. Wir erhalten schließlich als Ergebnis 00011000, die Binärdarstellung der Dezimalzahl 24.

   Zur Simulation verwenden wir einen `type SimulNat is array (0..7) of Boolean;` – der Wert an Index `i` sei dabei genau dann `true`, wenn das $i$-te Bit der simulierten Zahl den Wert 1 hat.

   - (1 Punkt, leicht) Schreiben Sie eine Funktion, die als Eingabe eine 8-Bit-Zahl erhält und den zugehörigen Dezimalwert zurückgibt.
   - (2 Punkte, mittel) Schreiben Sie eine Funktion, die als Eingabe eine Dezimalzahl erhält und die zugehörige 8-Bit-Zahl zurückgibt.

     Hinweis: Ist die Zahl ungerade, so wird das Bit mit der Wertigkeit $2^0$ eine 1 sein. Die höherwertigen Stellen erhält man, indem man die Dezimalzahl halbiert und erneut testet, ob diese nun gerade oder ungerade ist. Beispiel: Die Zahl 13 ist ungerade, das Bit 0 muss 1 sein. Ganzzahldivision der 13 durch 2 ergibt 6 – 6 ist gerade, das Bit 1 ist also 0. Ganzzahldivision der 6 durch 2 ergibt 3 – 3 ist ungerade, das Bit 2 ist demnach 1. Ganzzahldivision der 3 durch 2 ergibt 1 – 1 ist ungerade, das Bit 3 ist ebenfalls 1. Ganzzahldivision der 1 durch 2 ergibt 0 – alle höherwertigen Bits sind 0. Als 8-Bit-Zahl erhält man somit die 00001101 – beachten Sie, dass das niedrigstwertige Bit ganz rechts steht (so wie auch bei den Dezimalzahlen die Einer ganz rechts stehen).
   - (2 Punkte, leicht–mittel) Schreiben Sie eine Prozedur `AddEins`, die auf eine gegebene Zahl in Binärdarstellung 1 addiert. Von welcher Art muss der Parameter sein? Wie erkennt man, falls die resultierende Zahl sich nicht mehr mit 8 Bit darstellen lässt?

- (2 Punkte, mittel) Schreiben Sie eine Funktion `Addiere`, die zwei Binärzahlen mit je 8 Bit als Eingabe erhält und als Ergebnis ebenfalls eine 8-Bit-Zahl zurückgibt. Erkennen Sie auch hier, falls ein Überlauf passiert.
- Will man negative Zahlen darstellen, so deutet man in der Regel das höchstwertige Bit (hier Bit 7) als $-2^7$. Dies ist die sogenannte Zweierkomplement-Darstellung. Der mit 8 Bit darstellbare Bereich ist nun -128 bis 127.

  Zu einer positiven 8-Bit-Zahl erhält man die zugehörige negative 8-Bit-Zahl, indem man alle Bits negiert und auf die so erhaltene Zahl 1 mit obigem Algorithmus addiert. Beispiel: Negiert man die 8-Bit-Darstellung von 22 (dies ist 00010110), so erhält man zunächst 11101001, Addition von 1 führt zu 11101010. Diese Zahl hat den Wert $-128 + 64 + 32 + 8 + 2 = -22$.

  (2 Punkte, mittel) Schreiben Sie eine Prozedur `Negiere`, die ein 8-Bit-Zahl mit Dezimalwert `x` in eine 8-Bit-Zahl mit Dezimalwert `-x` verwandelt.

  (2 Punkte, mittel–schwer) Fügen Sie als Kommentarzeilen mit ein, warum dies funktioniert. Bei welcher Zahl funktioniert das Verfahren nicht?
- **Zusatzaufgabe (6 Punkte, sehr schwer, nur für Tüftler):** Bei der Zweierkomplement-Darstellung gibt jeweils das höchstwertige Bit an, ob eine Zahl negativ ist.

  Eine auf den ersten Blick sehr merkwürdig anmutende Zahldarstellung ist die zur Basis „-2“. Die Binärzahl 1101 hat dann z.B. den Wert $1\cdot(-2)^3+1\cdot(-2)^2+0\cdot(-2)^1+1\cdot(-2)^0 = -8+4-1=-5$. Die Dezimalzahl +7 hätte die Binärdarstellung 11001 (16-8-1).

  Finden Sie Algorithmen zur Negierung und Addition solcher Zahlen, die zur Basis $-2$ dargestellt sind.

  Hinweis: Diese Zahldarstellung demonstriert lediglich, dass es Alternativen zur Zweierkomplement-Darstellung gibt. In der Praxis findet sie meines Wissens nach keinerlei Anwendung.

Demonstrieren Sie die von Ihnen implementierten Prozeduren und Funktionen durch geeignete Testfälle im Hauptprogramm.

3. (5 Punkte) **Permutationen:** Als Permutation bezeichnet man die Umordnung einer vorgegebenen Zahlenfolge. Beispiel: Die Permutationen zur Menge $\{1, 2, 3\}$ lauten (1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,1,2), (3,2,1). Zu einer Menge mit $n$ Elementen, gibt es stets $n!$ Permutationen (die erste Stelle ist eine von $n$ möglichen Zahlen, abhängig von der ersten Stelle können $n-1$ Zahlen für die zweite Stelle ausgewählt werden, usw. – es ergeben sich $n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 = n!$ Möglichkeiten).

   - (5 Punkte, mittel–schwer) Schreiben Sie eine Prozedur, die zum Parameter $n$ alle Permutation der Menge $\{1, 2, \ldots, n\}$ ausgibt. Schreiben Sie dazu eine rekursive Prozedur, die ein Array nach und nach füllt unter Beachtung, dass der bisherige Inhalt des Array stets zu einer Permutation ergänzt werden kann.
   - **Alternativ-Aufgabe (schwerer, 5 Punkte):** Finden Sie einen Algorithmus, der zu einer gegebenen Permutation, die lexikographisch nächstgrößere Permutation findet (in obigem Beispiel sind die Permutation lexikographisch geordnet). Benutzen Sie diesen, um nacheinander alle Permutationen der Menge $\{1, 2, \ldots, n\}$ auszugeben.

   Es wird nur eine der beiden Lösungen gewertet!

4. (3 Punkte) **`for`-Schleifen**: Wir haben gelernt, dass die Laufvariblen von `for`-Schleifen implizit deklariert werden. So gesehen stellt eine `for`-Schleife ebenfalls einen Block dar.

   (2 Punkte) Simulieren Sie eine `for`-Schleife durch einen Block und eine `while`-Schleife. Testen Sie Ihr Vorgehen an dem Beispiel:

```
for i in 1..10 loop
  for i in 1..10 loop
    put(i,0);
  end loop;
end loop;
```

   (1 Punkt) Fügen Sie als Kommentarzeilen neben Ihrer Idee hinzu, ob und (wenn ja) welche Eigenschaften der Laufvariablen von der Simulation nicht abgebildet werden.

5. **Zusatzaufgabe Programmanalyse (schwer, 2+6 Punkte):** Der Student Exit hat bei unserer Tutorin Goto das folgende unkommentierte Programm abgegeben.

```
with text_io, ada.Integer_Text_Io;
use text_io, ada.Integer_Text_Io;

procedure Exit_Considered_Harmful is
   Zahl1, Zahl2, Zahl3 : Integer := 0;
begin
   put("Erste Zahl: ");
   get(Zahl1);
   Put("Zweite Zahl: ");
   Get(Zahl2);
   for I in 1..Zahl1 loop
      Zahl2 := Zahl2 + 3;
      exit when Zahl2 > 100;
      for J in Zahl1..Zahl1+10 loop
         exit when Zahl1 > Zahl2;
         Zahl2 := Zahl2 - Zahl1;
         if I+15 > J then
            Zahl3 := Zahl3 + 1;
            exit when Zahl3 > Zahl1;
         end if;
         Zahl2 := Zahl2 + Zahl1 + 1;
      end loop;
   end loop;
   Put("Ergebnis: ");
   Put(Zahl1-Zahl3);
   Put(Zahl2);
end Exit_Considered_Harmful;
```

Beide stehen vor einem Rätsel, was das Programm tatsächlich tut. Da sich unter Ihnen Spezialisten für die Transformation von bestehenden Programmen und die Reformulierung von Schleifen befinden, transformieren Sie das Programm in eine Version `ohneexit.adb`, die keine `exit`-Anweisungen mehr enthält und sich identisch zum Ursprungsprogramm verhält (2 Punkte). Vereinfachen Sie das Programm und versehen Sie es mit sinnvollen Kommentaren, sodass die Verständlichkeit erhöht wird (bis zu 6 Punkte). Sie können davon ausgehen, dass die Eingaben für das Programm immer zwischen 0 und 200 liegen. Das obige Programm ist auf der Webseite der Vorlesung erhältlich.

Hinweis1: Die Überführung in ein Programm ohne `exit` ist relativ einfach. Um das Programm jedoch in eine leicht lesbar, klar verständliche Form zu bringen, ist vermutlich ein wesentlich größerer Aufwand notwendig. Wägen Sie ab, wieviel Zeit Sie investieren möchten.

Hinweis2: Falls Sie die Anweisung `exit when` nicht kennen: Es handelt sich dabei um eine Sprunganweisung, welche die aktuelle (innerste) Schleife sofort verlässt, falls die Bedingung hinter `when` erfüllt ist. Die Programmausführung wird direkt hinter der Schleife fortgesetzt.

---