

Programmierübungen

Wintersemester 2007/2008

6. Übungsblatt

8. Dezember 2007

Abgabe bis Freitag, 14. Dezember 23:59 Uhr

Die Abgabe Ihrer Bearbeitung können Sie im eClaus-System durchführen. Erarbeiten Sie Lösungsideen zu den Aufgaben möglichst in Kleingruppen. Es wird jedoch von Ihnen erwartet, dass jeder Teilnehmer eine eigene Lösung abgibt. Sollten kopierte Quelltexte abgegeben werden, so werden grundsätzlich alle Kopien mit 0 Punkten bewertet. In den Vortragsfolien der Programmierübungen oder im Skript zur Einführung in die Informatik abgedruckte Quelltexte können verwendet werden, müssen aber der Programmierrichtlinie entsprechend formatiert und kommentiert werden.

Beachten Sie die Programmierrichtlinie und kommentieren Sie Ihren Quelltext! Dokumentieren Sie unbedingt Ihre Lösungsidee in den Quelltext-Kommentaren.

<http://www.iste.uni-stuttgart.de/ps/Lehre/WS0708/inf-prokurs>

Aufgabe 6.1: Abstrakter Datentyp

(10 Punkte)

Ein abstrakter Datentyp (ADT) ist ein Typ dessen sichtbare Eigenschaften ausschließlich durch eine Menge von Unterprogrammen definiert sind. Diese Unterprogramme werden als die Schnittstelle des ADT bezeichnet. Sie besitzen formale Parameter des ADT, oder der Typ ihres Rückgabewerts ist der ADT. Programme, die den ADT verwenden, brauchen nur dessen Schnittstelle zu kennen, nicht jedoch seine konkrete Darstellung oder die Implementierung der Unterprogramme.

Ein ADT wird in Ada innerhalb einer Paket-Spezifikation als privater Typ deklariert. Programmeinheiten, die das Paket mit einer **with** clause einbinden, können Variablen des Typs deklarieren, Zuweisungen an diese Variablen ausführen und den Gleichheitsoperator = auf zwei Werte des Typs anwenden. Weitere Funktionalität muss durch die Unterprogramme des Pakets angeboten werden (Schnittstelle des ADT).

Auf der Webseite zu den Programmierübungen erhalten Sie die Spezifikation eines Pakets Fractions. In diesem Paket wird der ADT Fraction als privater Typ deklariert.

```
type Fraction is private;
```

Im privaten Teil der Paketspezifikation (eingeleitet durch die Zeile, in der nur das Schlüsselwort **private** steht) ist die tatsächliche Darstellung des Typs als Record angegeben. Diese Deklaration ist in der Implementierung des Pakets sichtbar, nicht jedoch in einem Hauptprogramm, welches das Paket Fractions verwendet.

```
type Fraction is
  record
    Numerator   : Integer := 0;
    Denominator : Positive := 1;
  end record;
```

Der Typ Fraction soll einen mathematischen Bruch modellieren. Die Komponente Numerator speichert den Zähler des Bruchs, Denominator den Nenner.

1. Erstellen Sie eine Implementierung des Pakets Fractions, passend zu der Paket-Spezifikation. Achten Sie in Ihrem Quelltext besonders auf Vorzeichen und vermeiden Sie soweit möglich Überläufe in Ihren Berechnungen.
2. Schreiben Sie ein Hauptprogramm „Calculator“, das Fractions verwendet, um einen einfachen Taschenrechner zu simulieren. Der Taschenrechner arbeitet folgendermaßen:
 - a) Der Taschenrechner fordert die Benutzerin zur Eingabe eines ersten Bruchs auf und liest diesen Bruch ein. Dieser Bruch ist der erste Operand a .
 - b) Der Taschenrechner fordert die Benutzerin zur Eingabe einer der vier Grundrechenarten oder des Gleichheitszeichens auf und liest die Eingabe \circ (eines der Zeichen $+$, $-$, $*$, $/$ oder $=$).
 - c) Wurde das Zeichen $=$ eingegeben, beendet sich das Programm.
 - d) Der Taschenrechner fordert die Benutzerin zur Eingabe eines Bruchs auf und liest diesen ein. Dieser Bruch ist der zweite Operand b .
 - e) Der Taschenrechner berechnet das Resultat $r := a \circ b$ und gibt es aus. Dann setzt der Taschenrechner das Resultat als ersten Operanden ein $a := r$ und fährt mit Schritt 2b fort.

Hinweise:

- Beachten Sie, dass Ada implizit den Operator $=$ für jeden privaten Typ deklariert. Damit dieser Operator korrekt funktioniert, muss jeder Bruch normalisiert werden. D. h. jeder Bruch muss vollständig gekürzt sein, nur der Zähler eines Bruchs darf negativ werden, der Nenner jedoch nicht, und die 0 wird immer mit Nenner 1 dargestellt.
- In dem Paket Fraction_Helpers, das ebenfalls auf der Webseite erhältlich ist, werden die aus den früheren Übungen bereits bekannten Unterprogramme LCM und GCD zur Verfügung gestellt. Sie können sie benutzen oder wahlweise durch Ihre eigene Implementierung ersetzen.
- Um im Hauptprogramm die Operatoren aus dem Paket Fractions zu verwenden, können Sie (neben den anderen Möglichkeiten) eine use type clause einsetzen:

```
use type Fractions.Fraction;
```

Aufgabe 6.2: Einfach verkettete Listen

(10 Punkte)

Im Skript zur Vorlesung „Einführung in die Informatik 1“ wird in Abschnitt 3.5.2.0 ein Rahmenprogramm für die Behandlung von einfach verketteten Listen angegeben. Setzen Sie diesen Rahmen (das Programm Anwendung) in ein eigenes Programm um. Verwenden Sie als Typ Item den Typ Integer. Nennen Sie Ihr Programm Integer_Liste. Sie müssen in Ihr Programm nur die Operationen einfügen, die im folgenden tatsächlich verwendet werden. Achten Sie darauf, dass der Speicher für Listenzellen, die nicht mehr über Zeiger erreichbar sind, freigegeben wird.

Erweitern Sie das Programm so, dass als Kommandozeilen-Argument der Name einer Textdatei übergeben werden kann. Die Textdatei enthält nur ganze Zahlen, getrennt durch Leerzeichen. Das Programm liest die Textdatei und speichert alle Werte in einer Liste.

Implementieren Sie folgende weitere Operationen:

- Umkehren der Reihenfolge der Listeneinträge,
- Sortieren der Liste.
- Zerstören der Liste. Hier muss besonders darauf geachtet werden, dass der für die Listenzellen allokierte Speicher wieder freigegeben wird.

Diese Operationen sollen ausschließlich durch Zeiger-Zuweisungen implementiert sein. Der Integer-Inhalt einer Listenzelle darf nach dem Einlesen der Liste nicht mehr verändert werden.

Lassen Sie das Programm dann verschiedene Ausgaben erzeugen:

1. Ausgabe der Original-Liste,
2. Ausgabe der Liste in umgekehrter Reihenfolge,
3. Ausgabe der sortierten Liste.

Beispiel (zahlen.txt):

```
3 4 1 6 89 -45
4 9 23 12
```

Testlauf des Programms:

```
$ integer_liste zahlen.txt
Originalliste:
3, 4, 1, 6, 89, -45, 4, 9, 23, 12
umgekehrte Liste:
12, 23, 9, 4, -45, 89, 6, 1, 4, 3
sortierte Liste:
-45, 1, 3, 4, 4, 6, 9, 12, 23, 89
```

Hinweise:

- Es reicht nicht aus, dass Ihr Programm die korrekte Ausgabe erzeugt. Die Listenoperationen müssen implementiert werden und im Quelltext erkennbar sein. Es ist nicht zulässig, die Operationen auf einem Array durchzuführen und daraus wieder eine neue Liste zu erzeugen.
- Speicher wird mit einer Instanz von `Ada.Unchecked_Deallocation` freigegeben. Ein Beispiel finden Sie in der Datei `pointer_example.adb`, das Sie auch von der Webseite herunter laden können.
- Zum Sortieren der Liste können Sie folgenden Algorithmus verwenden (Selection Sort): Es soll die Quell-Liste S sortiert werden. Dazu wird eine anfangs leere Zielliste T verwendet. Der Algorithmus führt solange Schritte durch, bis S die leere Liste geworden ist. In jedem Schritt wird aus S das jeweils größte Element entfernt und an den Anfang der Liste T hinzugefügt.

Aufgabe 6.3: Einsatz eines ADT

(bis zu 2 Zusatzpunkte)

Bitte beachten: Die Bearbeitung dieser Aufgabe ist freiwillig. Es können bis zu 2 Punkte erworben werden, jedoch kann die Anzahl erworbener Punkte für das ganze Aufgabenblatt nicht über 20 Punkte gesteigert werden.

Ergänzen Sie zu dem ADT Fraction einen Vergleichsoperator "<". Kopieren und ändern Sie die Implementierung Ihrer Listenimplementierung und des Hauptprogramms aus Aufgabe 6.2 zu `fraction_liste` so dass nun statt ganzen Zahlen Brüche verarbeitet werden können.

Bitte geben Sie auf jeden Fall trotzdem die ursprüngliche (unveränderte) Variante separat ab, da sich andernfalls die Korrektur erschwert. Es ist *nicht* möglich ausschließlich die Zusatzaufgabe zu bearbeiten.

Hinweis: Vergleichsoperatoren können in Ada-Programmen durch Funktionen angegeben werden (s. u.). Ist diese Funktion im Paket Fractions deklariert, so kann sie im Hauptprogramm durch `use type Fractions.Fraction;` sichtbar gemacht werden.

```
function "<"  
  (Left  : in Fraction;  
   Right : in Fraction)  
  return Boolean;
```