Fakultät IV: Elektrotechnik und Informatik
Informatik 1 WS 2001/02,
Jähnichen / Flik / Lechtchinsky / Neunast / Tutoren

Aufgabenblatt 3

letzte Aktualisierung: 21. November, 15:04

Ausgabe: 09.11.2001
Abgabe: 19./20.11.2001 Prozent: 100

Thema: *Rekursionen, Sequenzen*

Mit diesem Aufgabenblatt soll Euer Verständnis für rekursive Programmierung vertieft, und die Benutzung von Sequenzen in Opal eingeführt werden.

1. Aufgabe (Tut): Rekursionsarten

Welche Rekursionstypen gibt es, worin unterscheiden sie sich?

Lösung:

Neben repetitiver und linearer Rekursion gibt es noch baumartige Rekursion und geschachtelte Rekursion. Außerdem können diese Rekursionsarten sowohl direkt als auch indirekt sein.

Repetitive Rekursion ist die einfachste Art. In jedem Zweig der Fallunterscheidung (die für jede rekursive Funktion benötigt wird, evtl. durch Pattern-Matching realisiert) wird entweder nicht-rekursiv ein Ausdruck berechnet (Rekursionsanker); oder der rekursive Aufruf steht als äußerstes in dem zu berechnenden Ausdruck (und kommt darin nur einmal vor). Damit kann das durch die Rekursion berechnete Funktionsergebnis gleich als Ergebnis der betrachteten Rekursionsebene zurückgegeben werden.

Die lineare Rekursion sieht ähnlich aus. Allerdings kann nun der (weiterhin einzige) rekursive Aufruf innerhalb des berechnenden Ausdrucks stehen. Es wird also mit dem Ergebnis des rekursiven Aufrufs evtl. noch eine Berechnung angestellt, bevor deren Ergebnis als Ergebnis der Funktion zurückgeliefert wird.

Bei der baumartigen Rekursion gibt es in mindestens einem Zweig der Funktion nicht nur einen, sondern mindestens zwei rekursive Auswertungen nebeneinander; bei der geschachtelten Rekursion gibt es ebenfalls in mindestens einem Zweig mindestens zwei rekursive Auswertungen, allerdings nicht nebeneinander. Stattdessen wird eine rekursive Auswertung benutzt, um das Argument für den anderen rekursiven Aufruf zu berechnen.

Indirekte Rekursion bedeutet, dass die rekursive Funktion eine andere Funktion aufruft, die (evtl. über mehrere Zwischenschritte) sie selber wieder benutzt.

2. Aufgabe (50 Prozent): Sequenzen

Sequenzen sind eine für die funktionale Programmierung unverzichtbare Datenstruktur. Da sie inhärent rekursiv sind, lässt sich an Algorithmen, die mit ihnen arbeiten, rekursive Programmierung sehr gut üben. Sequenzen in Opal können verschiedene Datentypen als Element enthalten (wobei innerhalb einer Sequenz alle Elemente den gleichen Typ haben), deshalb muss man den gewünschten beim Import in "[]" angeben, z.B.:


```
IMPORT Nat ONLY nat
IMPORT Seq[nat] ONLY seq[nat]
```


Neben dem Datentyp `seq[`α`]` kann man aus der Struktur `Seq` noch Funktionen zum Umgang mit Sequenzen importieren. Wichtig sind für die Bearbeitung dieser Aufgabe die Funktionen `<>`, `::`, `<>?`, `::?`, `ft` und `rt` (verwendet für die Lösung dieser Aufgabe keine anderen aus der Struktur `Seq`!).

2.1. Sequenzen, Reduce und Fold, Akkumulatoren (Tut) Implementiert die Funktionen `reduceSum` und `foldDiff`, die alle Elemente einer Sequenz von `nat`s zu einem zusammenfassen. Als Operation für das Reduzieren jeweils zweier Elemente zu einem sollt Ihr "`+`" für das rechtsassoziative `reduceSum`, und "`-`" für das linksassoziative `foldDiff` verwenden. Welches Ergebnis liefert jeweils ein Aufruf `reduceSum(8 :: 1 :: 3 :: <>)` und `foldDiff(8 :: 1 :: 3 :: <>)`?

Lösung:

Ergebnisse:
`reduceSum(8 :: 1 :: 3 :: <>)` = 12
`foldDiff(8 :: 1 :: 3 :: <>)` = 4.

```
IMPORT Nat COMPLETELY
IMPORT Seq[nat] COMPLETELY

-- Addiert alle Zahlen in der übergebenen Sequenz "s"
FUN reduceSum: seq[nat] -> nat
DEF reduceSum == \\s.
    IF <>?(s) THEN "0"!
    IF ::?(s) THEN ft(s) + reduceSum(rt(s))
    FI

-- Subtrahiert die Elemente der Sequenz "s" voneinander
FUN foldDiff: seq[nat] -> nat
DEF foldDiff == \\s.
    IF <>?(s) THEN "0"!
    IF ::?(s) THEN foldDiffHelp(ft(s),rt(s))
    FI

-- Hilfsfunktion zu "foldDiff". "x" enthält das bisher berechnete Ergebnis.
FUN foldDiffHelp: nat ** seq[nat] -> nat
DEF foldDiffHelp == \\x,s.
    IF <>?(s) THEN x
    IF ::?(s) THEN foldDiffHelp(x-ft(s), rt(s))
    FI
```

2.2. Skalarprodukt Schreibt eine Funktion, die das Skalarprodukt zweier als Sequenzen repräsentierter Vektoren berechnet. Das Skalarprodukt zweier Vektor der gleichen Dimension ist definiert als die Summe der Produkte der jeweils korrespondierenden Elemente beider Vektoren. Zum Beispiel ist $(1,2,4)\cdot(2,1,3) = 1\cdot 2 + 2\cdot 1 + 4\cdot 3 = 16$.

Lösung:

```
-- Berechnet das Skalarprodukt der beiden Vektoren "s1" und "s2"
-- Die Sequenzen s1 und s2 müssen die gleiche Länge haben, ansonsten ist das
-- Verhalten von skalarProd undefiniert
FUN skalarProd: seq[real] ** seq[real] -> real
```

```
DEF skalarProd == \\s1,s2.
    IF (<>?(s1) and <>?(s2)) THEN 0
    IF (::?(s1) and ::?(s2)) THEN (ft(s1)*ft(s2)) + skalarProd(rt(s1), rt(s2))
    FI
```


2.3. Horner-Schema Schreibt eine Funktion, die ein Polynom unter Verwendung des Horner-Schemas an einer Stelle x auswertet. Die Koeffizienten des Polynoms sollen der Funktion als Sequenz übergeben werden. Ein Beispiel: Das Polynom $2x^3 - 5x + 3$ sieht als Sequenz so aus: "`2 :: 0 :: -5 :: 3 :: <>`". Um es an der Stelle x auszuwerten, berechnet man das folgende: $(((2) \cdot x + 0) \cdot x - 5) \cdot x + 3$.

Lösung:

```
-- Wertet das Polynom, dessen Koeffizienten in der Sequenz "s" übergeben
-- werden, mit Hilfe des Horner-Schemas an der Stelle "x" aus.
FUN horner: seq[real] ** real -> real
DEF horner == \\s,x. hornerHelp(s,x,0)

FUN hornerHelp: seq[real] ** real ** real -> real
DEF hornerHelp == \\s,x,acc.
    IF (<>?(s)) THEN acc
    IF (::?(s)) THEN hornerHelp(rt(s), x, (acc*x)+ft(s))
    FI
```

2.4. Liste von 1 bis n generieren Schreibt eine Funktion, die eine Zahl n übergeben bekommt und als Ergebnis eine Sequenz liefert, die die natürlichen Zahlen von 1 bis n enthält (beide inklusive). Wenn 0 übergeben wird, soll die leere Sequenz zurückgegeben werden.

Lösung:

```
-- Generiert eine Sequenz mit den Zahlen von 1 bis n
FUN genSeq: nat -> seq[nat]
DEF genSeq == \\n. genSeqHelp(<>,n)

FUN genSeqHelp: seq[nat] ** nat -> seq[nat]
DEF genSeqHelp == \\s,n.
    IF (n=0) THEN s
    IF (n>0) THEN genSeqHelp((n :: s),(n-1))
    FI
```

oder alternativ

```
-- Generiert eine Sequenz mit den Zahlen von 1 bis n
FUN genSeq2: nat -> seq[nat]
DEF genSeq2 == \\n. genSeqHelp2(1,n)

FUN genSeqHelp2: nat ** nat -> seq[nat]
DEF genSeqHelp2 == \\i,n.
    IF (i>n)  THEN <>
    IF (i<=n) THEN i :: genSeqHelp2((i+1),n)
    FI
```

3. Aufgabe (50 Prozent): Sortieren

Zur Lösung dieser Aufgabe sind neben den bereits vorgestellten Funktionen noch weitere aus der Struktur `Seq` sehr hilfreich. Schaut Euch deshalb die Dokumentation zu dieser Struktur gut an!

Diese Aufgaben sollen ohne Zuhilfenahme von Opal-Funktionen wie `filter`, `reduce`, `fold`, `map`, `zip`, u.ä. erledigt werden.

3.1. MergeSort (Tut) Implementiert den Sortieralgorithmus Mergesort. Die Funktion `mergeSort` bekommt eine zu sortierende Sequenz natürlicher Zahlen übergeben und liefert eine entsprechend sortierte Sequenz zurück.

Lösung:

```
IMPLEMENTATION MergeSort

IMPORT Nat COMPLETELY
IMPORT Seq[nat] COMPLETELY

-- Unsortierte Sequenz zu Testzwecken
FUN testSeq : seq[nat]
DEF testSeq == %(5,7,6,4,8,3,9,2) ++ %(0,1,10)

-- Sortiert die übergebene Sequenz mit Hilfe des Sortier-
-- verfahrens MergeSort
FUN mergeSort : seq[nat] -> seq[nat]
DEF mergeSort == \\ s .
        IF <>?(s) THEN <>
        -- leere Liste sortieren
        OTHERWISE
        IF <>?(rt(s)) THEN ft(s) :: <>
        -- einelementige Liste sortieren
        ELSE    LET (s1,s2) == split(s)
                IN  merge(mergeSort(s1),mergeSort(s2))
                -- andere Listen sortieren
        FI

-- Splittet eine Liste in zwei Listen
FUN split : seq[nat] -> seq[nat] ** seq[nat]
DEF split(<>) ==
        (<>, <>)
DEF split(x::xs) ==
        LET (s1, s2) == split(xs)
        IN  (s2, x::s1)

-- Fuegt zwei sortierte Listen zusammen
FUN merge : seq[nat] ** seq[nat] -> seq[nat]
DEF merge == \\ s1,s2 .
        IF <>?(s1) THEN s2
        IF <>?(s2) THEN s1
        OTHERWISE
        IF ft(s1) <= ft(s2) THEN ft(s1) :: merge(rt(s1),s2)
        IF ft(s2) <= ft(s1) THEN ft(s2) :: merge(s1,rt(s2))
        FI
```

3.2. QuickSort Implementiert den Sortieralgorithmus Quicksort in der Funktion `quickSort`, die die gleiche Funktionalität hat wie `mergeSort`.

QuickSort ist ein sehr oft benutztes Sortierverfahren, da es eine einfache Implementation mit (meistens) hoher Sortiergeschwindigkeit verknüpft. Eine übergebene Sequenz wird dabei wie folgt sortiert:

- Wenn die Sequenz leer ist, ist sie schon sortiert.
- Sonst wähle ein Element (genannt Trennelement) der Sequenz aus und teile die anderen Elemente in zwei Sequenzen auf; die eine soll Elemente kleiner dem Trennelement enthalten, und die andere Elemente größer oder gleich dem Trennelement.
- Nun sortiere rekursiv die beiden Teilsequenzen.
- Die sortierte Sequenz ergibt sich aus dem Zusammenfügen der sortierten Teilsequenz mit den kleineren Elementen, dem Trennelement, und der sortierten Teilsequenz mit den größeren/gleichen Elementen.

Lösung:

```
IMPLEMENTATION QuickSort

IMPORT Nat COMPLETELY
IMPORT Seq[nat] COMPLETELY

-- Sortiert eine übergebene seq[nat] mit Hilfe von QuickSort
FUN quickSort : seq[nat] -> seq[nat]
DEF quickSort(<>) ==
        <>
DEF quickSort(el::rest) ==
        -- "+%+" fügt el zwischen den beiden Argument-Sequenzen ein
        +%+(el)(quickSort(smaller), quickSort(greater))
        WHERE   smaller == filterSmaller(rest, el)
                greater == filterGreaterEq(rest, el)

-- Hilfsfunktion für quickSort: Filtert alle Elemente einer Sequence,
-- die kleiner als ein übergebener Wert sind
FUN filterSmaller : seq[nat] ** nat -> seq[nat]
DEF filterSmaller(<>, val) ==
        <>
DEF filterSmaller(el::rest, val) ==
        IF (el < val)  THEN el :: filterSmaller(rest, val)
        IF (el >= val) THEN       filterSmaller(rest, val)
        FI

-- Hilfsfunktion für quickSort: Filtert alle Elemente einer Sequence,
-- die größer/gleich einem übergebenen Wert sind
FUN filterGreaterEq : seq[nat] ** nat -> seq[nat]
DEF filterGreaterEq(<>, val) ==
        <>
DEF filterGreaterEq(el::rest, val) ==
        IF (el >= val) THEN el :: filterGreaterEq(rest, val)
        IF (el < val)  THEN       filterGreaterEq(rest, val)
        FI
```