Ludwig-Maximilians-Universität München
Department „Institut für Informatik“
Prof. Dr. Christian Böhm
Elke Achtert, Arthur Zimek

München, 12.01.2007

Informatik 1
WS 2006/07

Übungsblatt 11: (Rekursive) Datentypen, Strukturelle Induktion

Besprechung: 22.01.–26.01.2007

Abgabe aller mit **Hausaufgabe** markierten Aufgaben bis Freitag, 19.01.2007, 18:00 Uhr

Aufgabe 11-1 *Ein Rätsel zur Wiederholung* **(Hausaufgabe)**

Sicher kennen Sie den Rätseltyp „Sudoku“, ein Logik-Rätsel, das aus einem Gitterfeld mit 3×3 Blöcken besteht, die jeweils in 3×3 Felder unterteilt sind. Insgesamt gibt es also 81 Felder in 9 Reihen und 9 Spalten. In einige dieser Felder sind schon zu Beginn Ziffern zwischen 1 und 9 eingetragen. Typischerweise sind 22 bis 36 Felder von 81 möglichen vorgegeben. Ziel des Spiels ist es nun, die leeren Felder so mit Ziffern zwischen 1 und 9 auszufüllen, dass in jeder der je neun Zeilen, Spalten und Blöcke jede Ziffer von 1 bis 9 genau einmal auftritt.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

(a) Um ein beliebiges Sudoku durch SML lösen zu können, wollen wir zunächst einige spezielle Datentypen erstellen.

- Der Typ `ziffer` hat neun Werte, die sich – nach Definition des Datentyps – auflisten lassen wie folgt:

```
- val ziffern = [z1 , z2 , z3 , z4 , z5 , z6 , z7 , z8 , z9];
val ziffern = [z1,z2,z3,z4,z5,z6,z7,z8,z9] : ziffer list
```

- Der Typ `symbol` kann aus dem Wert `Leer` bestehen, oder aus einem Symbol für eine Ziffer, hat also einen 0-stelligen Konstruktor `Leer` und einen 1-stelligen Konstruktor `Symbol` vom Typ `ziffer -> symbol`.
- Der Datentyp `sudoku` hat einen einstelligen Konstruktor `Sudoku` vom Typ `symbol list -> sudoku`.

Geben Sie die hierfür benötigten Definitionen in einer Datei `11-1.sml` an.

(b) Um ein Sudoku relativ einfach einzugeben, wollen wir es auch aus einem String erzeugen können, der z.B. aussehen könnte wie folgt:

```
val sudokuString = "53  7    |\n\
                   \6  195   |\n\
                   \ 98    6 |\n\
                   \8   6   3|\n\
                   \4  8 3  1|\n\
                   \7   2   6|\n\
                   \ 6    28 |\n\
                   \   419  5|\n\
                   \    8  79|\n";
```

Dieser String soll das Sudoku aus der Illustration oben darstellen.
Eine leere Stelle wird durch das Zeichen `#" "` markiert.
Die Ziffern 1-9 werden durch die Zeichen `#"1"` bis `#"9"` markiert.
Das Zeichen `#"|"` markiert ein Zeilenende (zur Sicherheit, da Leerzeichen am Zeilenende bei vielen Editoren nicht mitgespeichert werden).
Schließlich gibt es noch den Zeilenumbruch, markiert durch das Zeichen `#"\n"`.
Den gleichen String, wie er in einer Textdatei angegeben würde, finden Sie in der Datei `sudoku.txt`.
Nun benötigen wir eine Funktion `erzeugeSudoku` vom Typ `string -> sudoku`, die zu einem korrekt formatierten String wie oben (9 Zeilen mit je 9 Zeichen gefolgt von `"|\n"`) einen Wert vom Typ `sudoku` erzeugt.
Die Funktion `explode` könnte dazu hilfreich sein.
Definieren Sie eine solche Funktion in der gleichen Datei `11-1.sml`.

Hinweis: Wenn Sie diese Funktion definiert haben, können Sie folgende Funktion verwenden, um ein Sudoku aus einer korrekt formatierten Datei zu erzeugen:

```
fun sudokuEinlesen(file) = erzeugeSudoku(TextIO.inputAll(TextIO.openIn(file)));
```

Als Parameter erwartet diese Funktion einen Dateinamen (so wie von der Funktion `use` bekannt).

(c) Zur leichteren Lesbarkeit eines Sudokus brauchen wir nun auch die umgekehrte Richtung: Aus einem Wert vom Typ `sudoku` wollen wir einen String erzeugen. Der String soll genauso formatiert sein, wie der Eingabe-String für die Funktion `erzeugeSudoku` aus der vorherigen Teilaufgabe. Definieren Sie hierzu in der Datei `11-1.sml` eine Funktion `toString` vom Typ `sudoku -> string`.

(d) Wir benötigen nun eine Funktion, die für ein gegebenes Sudoku entscheidet, ob es gültig ist. Ein Sudoku ist gültig, wenn in jeder Zeile, in jeder Spalte und in jedem Block jede Ziffer nur in einem Symbol vorkommt. Das Symbol `Leer` darf jedoch beliebig oft vorkommen.
Definieren Sie hierfür in der Datei `11-1.sml` eine Funktion `gueltig` vom Typ `sudoku -> bool`.

Hinweis: Definieren Sie zunächst Hilfsfunktionen, die eine bestimmte Untereinheit eines Sudoku liefern, z.B. den dritten Block oder die achte Zeile. Liefern Sie diese Untereinheiten als Liste (vom Typ `symbol list`) zurück, dann können Sie hilfreiche Listenfunktionen aus der Vorlesung verwenden, um die Gültigkeit dieser Untereinheit zu überprüfen.
Wenn Ihnen die gesamte Aufgabe zu schwierig erscheint, versuchen Sie, kleine Schritte zu schaffen. Auch Teillösungen können Sie selbstverständlich abgeben. Es ist eine wichtige Fähigkeit für Informatiker, sinnvolle Teilprobleme zu identifizieren. Diese Fähigkeit können Sie hier üben.

(e) Wenn Sie in den bisherigen Teilaufgaben erfolgreich waren, können Sie nun eine Funktion zur Lösung eines Sudokus implementieren. Definieren Sie die Funktion `loese` vom Typ `sudoku -> sudoku` in der Datei `11-1.sml`. Die Funktion soll ein Sudoku auf ein vollständiges (d.h. ohne `Leer`-Symbole) und gültiges Sudoku abbilden, wenn es eine Lösung gibt, auf den Wert `Sudoku(nil)`, wenn es keine Lösung gibt. (Unser Beispiel-Sudoku hat eine Lösung.)

Hinweis: Eine einfache (und mit den Mitteln der Rekursion auch sehr einfach zu implementierende) Lösungsstrategie ist das sogenannte *Backtracking*:
Für das erste freie Feld versucht man das Symbol mit der ersten Ziffer einzufügen. Ergibt sich dadurch ein gültiges Sudoku, versucht man mit dem restlichen Sudoku ebenso zu verfahren. (Hierbei muß man sich natürlich den Anfang merken, damit man immer ein vollständiges Sudoku überprüfen kann.)
Andernfalls versucht man das Symbol mit der nächsten Ziffer einzufügen, man geht also bei einem falschen Lösungsweg soweit zurück (daher der Name *backtracking*), bis man vor einer Verzweigung steht, an der noch mehrere Möglichkeiten ungetestet sind.
Hat man keine Ziffern mehr übrig, so ist das Sudoku nicht lösbar.
Hat das Sudoku keine leeren Felder mehr, und ist es gültig, so ist es gelöst.

Aufgabe 11-2 *Rekursive Datentypen* **(Hausaufgabe)**

```
datatype formula = constant of int
                 | variable of string
                 | sum      of formula * formula
                 | product  of formula * formula;
```

Die Datei `11-2.sml` enthält die Definition dieses Datentyps, dessen Werte zur Repräsentation von arithmetischen Formeln dienen sollen. Der SML-Ausdruck `sum(constant 2, variable "x")` ist zum Beispiel ein Wert diese Typs, der die arithmetische Formel $2 + x$ repräsentiert.

(a) Definieren Sie in der selben Datei eine Funktion `toString` vom Typ `formula -> string` zur Erzeugung einer lesbareren, voll geklammerten Darstellung:

```
- toString(sum(constant 2, variable "x"));
val it = "(2 + x)" : string
- toString(product(constant 2, variable "x"));
val it = "(2 * x)" : string
- toString(sum(product(constant 2, variable "x"),
               product(variable "x", variable "x")));
val it = "((2 * x) + (x * x))" : string
```

Hinweis: Zur Konvertierung von `int` nach `string` dient die Funktion `Int.toString`.

(b) Definieren Sie eine Funktion `deriv` vom Typ `formula * formula -> formula` zur Berechnung der Ableitung (englisch *derivation*) einer Formel F nach einer Variablen x. Der zweite Parameter von `deriv` ist zwar vom Typ `formula`, aber die Funktion braucht nur die Fälle zu behandeln, in denen der zweite Parameter mit dem Konstruktor `variable` gebildet ist.

Für arithmetische Formeln gelten bekanntlich folgende Ableitungsregeln:

$\frac{d}{dx}c = 0$ für Konstante c $\qquad$ $\frac{d}{dx}(u+v) = \frac{d}{dx}u + \frac{d}{dx}v$

$\frac{d}{dx}v = 0$ für Variable $v \neq x$ $\qquad$ $\frac{d}{dx}(u \cdot v) \;= u \cdot (\frac{d}{dx}v) + (\frac{d}{dx}u) \cdot v$

$\frac{d}{dx}x = 1$

Beispiele:

```
- deriv( sum(constant 2, variable "x"),
         variable "x" );
val it = sum (constant 0,constant 1) : formula
- toString(deriv( product(constant 2, variable "x"),
                  variable "x" ));
val it = "((2 * 1) + (0 * x))" : string
- toString(deriv( sum(product(variable "x",variable "y"),
                      product(variable "x",variable "x")),
                  variable "x" ));
val it = "(((x * 0) + (1 * y)) + ((x * 1) + (1 * x)))" : string
- toString(deriv( sum(product(variable "x",variable "y"),
                      product(variable "x",variable "x")),
                  variable "y" ));
val it = "(((x * 1) + (0 * y)) + ((x * 0) + (0 * x)))" : string
```

(c) Die mit den obigen Ableitungsregeln erzeugten Formeln sind zwar mathematisch korrekt, aber komplizierter als man sie selbst schreiben würde. Sie können nach offensichtlichen Regeln simplifiziert werden:

```
- simpl(deriv( sum(constant 2, variable "x"),
               variable "x" ));
val it = constant 1 : formula
- toString(simpl(deriv( product(constant 2, variable "x"),
                        variable "x" )));
val it = "2" : string
- toString(simpl(deriv( sum(product(variable "x",variable "y"),
                            product(variable "x",variable "x")),
                        variable "x" )));
val it = "(y + (x + x))" : string
- toString(simpl(deriv( sum(product(variable "x",variable "y"),
                            product(variable "x",variable "x")),
                        variable "y" )));
val it = "x" : string
```

Zu den Simplifikationsregeln gehören: Eine Summe von zwei Konstanten kann zu einer Konstanten simplifiziert werden, die durch Addition der konstanten Summanden berechnet wird. Eine Summe, deren einer Summand 0 ist, kann zum anderen Summanden simplifiziert werden. Ein Produkt von zwei Konstanten kann zu einer Konstanten simplifiziert werden, die durch Multiplikation der konstanten Faktoren berechnet wird. Ein Produkt, dessen einer Faktor 0 ist, kann zur Konstanten 0 simplifiziert werden. Ein Produkt, dessen einer Faktor 1 ist, kann zum anderen Faktor simplifiziert werden.

Definieren Sie eine Funktion `simpl` vom Typ `formula -> formula` zur Simplifikation einer Formel nach diesen Regeln (ebenfalls in der selben Datei).

Aufgabe 11-3 *Strukturelle Induktion* **(Hausaufgabe)**

```
datatype formula = constant of int
                 | variable of string
                 | sum      of formula * formula
                 | product  of formula * formula;
```

Gegeben sei dieser Datentyp zur Repräsentation von arithmetischen Formeln. Die Konstruktoren `constant` und `variable` heißen *atomar*, die Konstruktoren `sum` und `product` heißen *binär*.

Geben Sie in einer Datei `11-3.txt` einen Beweis an, dass für jede Formel F vom Typ `formula` gilt: ist a die Anzahl der Vorkommen von atomaren Konstruktoren in F und b die Anzahl der Vorkommen von binären Konstruktoren in F, so ist $a = b + 1$.