

Programmierung und Modellierung, SS 2008  
Übungsblatt 9

Abgabe: bis Di 24.06.2008 (nachts)

Besprechung Di-Do 26.-30.06.2008

Aufgabe 9-1 SML

Datentypen

Betrachten Sie die Datei 9-1.sml.

- a) Ergänzen Sie die Definition des Datentyps `fach`. Das ist ein Aufzählungstyp mit den Werten `Informatik`, `Bioinformatik`, `Medieninformatik`, und `Sonstige`.
- b) Ergänzen Sie die Definition des Datentyps `student`. Er hat einen einzigen Konstruktor, der `Student` heißt. Der Typ muss zu den in der Datei vorhandenen Beispieldaten passen.

Zu jedem Typ `'a` ist `'a option` ein Typ bestehend aus den Werten `NONE` und `SOME v` für `v` vom Typ `'a`. Damit kann man jeden Typ `'a` um den Fall „kein Wert“ erweitern. Für diesen Typ sind folgende Funktionen in SML verfügbar:

`isSome opt`

liefert `false` wenn `opt` der Wert `NONE` ist; liefert `true` wenn `opt` ein Wert `SOME v` ist.

`valOf opt`

löst einen Fehler aus, wenn `opt` der Wert `NONE` ist; liefert `v` wenn `opt` ein Wert `SOME v` ist.

`getOpt (opt, a)`

liefert `a` wenn `opt` der Wert `NONE` ist; liefert `v` wenn `opt` ein Wert `SOME v` ist.

- c) Definieren Sie `mittelwert_punkte_zugelassen : student list -> real`. Die Funktion berechnet den Mittelwert der Punktzahlen, wobei der Wert `NONE` so behandelt wird wie der Wert `SOME 0`.
- d) Definieren Sie `mittelwert_punkte_teilgenommen : student list -> real`. Die Funktion berechnet den Mittelwert der Punktzahlen, wobei der Wert `NONE` so behandelt wird als käme der `student`-Datensatz gar nicht in der Liste vor.

Aufgabe 9-2 SML

Rekursive Datentypen

```
datatype formula = const of int
                  | var   of string
                  | sum   of formula * formula
                  | prod  of formula * formula;
```

Die Datei 9-2.sml enthält die Definition dieses Datentyps, dessen Werte zur Repräsentation von arithmetischen Formeln dienen sollen. Der SML-Ausdruck `sum(const 2, var "x")` ist zum Beispiel ein Wert dieses 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(const 2, var "x"));
val it = "(2 + x)" : string
- toString(prod(const 2, var "x"));
val it = "(2 * x)" : string
- toString(sum(prod(const 2, var "x"),
                prod(var "x", var "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 * string -> formula` zur Berechnung der Ableitung (englisch *derivation*) einer Formel  $F$  nach einer Variablen  $x$ .

Für arithmetische Formeln gelten bekanntlich folgende Ableitungsregeln:

$$\begin{array}{ll} \frac{d}{dx}c = 0 & \text{für Konstante } c \\ \frac{d}{dx}v = 0 & \text{für Variable } v \neq x \\ \frac{d}{dx}x = 1 & \end{array} \qquad \begin{array}{l} \frac{d}{dx}(u + v) = \frac{d}{dx}u + \frac{d}{dx}v \\ \frac{d}{dx}(u \cdot v) = u \cdot \left(\frac{d}{dx}v\right) + \left(\frac{d}{dx}u\right) \cdot v \end{array}$$

Beispiele:

```
- deriv( sum(const 2, var "x"), "x" );
val it = sum (const 0,const 1) : formula
- toString(deriv( prod(const 2, var "x"), "x" ));
val it = "((2 * 1) + (0 * x))" : string
- toString(deriv( sum(prod(var "x",var "y"),
                        prod(var "x",var "x")), "x" ));
val it = "(((x * 0) + (1 * y)) + ((x * 1) + (1 * x)))" : string
- toString(deriv( sum(prod(var "x",var "y"),
                        prod(var "x",var "x")), "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(const 2, var "x"), "x" ));
val it = const 1 : formula
- toString(simpl(deriv( prod(const 2, var "x"), "x" )));
val it = "2" : string
- toString(simpl(deriv( sum(prod(var "x",var "y"),
                        prod(var "x",var "x")), "x" )));
val it = "(y + (x + x))" : string
- toString(simpl(deriv( sum(prod(var "x",var "y"),
                        prod(var "x",var "x")), "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).

d) Eine Umgebung (englisch *environment*) sei eine Liste von Paaren wie zum Beispiel:

```
[(var "y",2), (var "x",4), (var "z",3)]
[(var "x",1), (var "y",2), (var "x",4), (var "z",3)]
```

Die erste dieser beiden Umgebungen repräsentiert, dass die Variable  $x$  an den Wert 4 gebunden ist, die Variable  $y$  an den Wert 2 und die Variable  $z$  an den Wert 3. Die zweite Umgebung repräsentiert, dass die Variable  $x$  an den Wert 1 gebunden ist, wodurch die Bindung von  $x$  an den Wert 4 überschattet wird. Die Bindungen von  $y$  und  $z$  sind wie in der ersten Umgebung.

Definieren Sie eine Funktion `value` vom Typ `formula * (formula * 'a) list -> 'a` die den Wert einer Variablen in einer Umgebung liefert, wie in den folgenden Beispielen:

```
- value( var "x", [(var "y",2), (var "x",4), (var "z",3)] );
val it = 4 : int
- value( var "x", [(var "x",1), (var "y",2),
                    (var "x",4), (var "z",3)] );
val it = 1 : int
- value( var "y", [(var "x",1), (var "y",2),
                    (var "x",4), (var "z",3)] );
val it = 2 : int
```

Falls für eine Variable der Formel kein Wert in der Umgebung vorhanden ist, soll `value` eine Ausnahme werfen.

e) Definieren Sie (nach wie vor in der selben Datei wie alles Bisherige) eine Funktion `eval` vom Typ `formula * (formula * int) list -> int` zur Auswertung einer arithmetischen Formel in einer Umgebung. Beispiele:

```
- eval( var "x", [(var "x",1), (var "y",2),
                  (var "x",4), (var "z",3)] );
val it = 1 : int
- eval( const 7, [(var "x",1), (var "y",2),
                    (var "x",4), (var "z",3)] );
val it = 7 : int
- eval( prod(sum(var "x",const 7), var "y"),
        [(var "x",1), (var "y",2),
         (var "x",4), (var "z",3)] );
val it = 16 : int
```

### Aufgabe 9-3

#### Strukturelle Induktion

```
datatype formula = const of int
                  | var   of string
                  | sum   of formula * formula
                  | prod  of formula * formula;
```

Gegeben sei dieser Datentyp zur Repräsentation von arithmetischen Formeln. Die Konstruktoren `const` und `var` heißen *atomar*, die Konstruktoren `sum` und `prod` heißen *binär*.

Geben Sie 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  $a = b + 1$ .