11. Übungsblatt zu Informatik 1 WS 98/99

Andreas Rossberg, Prof. Gert Smolka

Abgabe: 29. Januar 1999 vor der Vorlesung

Hinweis: Alle hier angegebenen Signaturen finden Sie unter `http://www.ps.uni-sb.de/courses/info1-ws98/uebung/sml/blatt11/`.

Bitte alle Programme per Email abgeben.

Aufgabe 11.1: Zahlenkonversion (6 Punkte)

(a) Die vorgegebene Prozedur `toString` aus dem `Bignat`-Modul arbeitet nur mit der Basis 10^4 für die Repräsentation der Zahlen. Geben Sie eine andere Implementierung an, die für beliebige Basen funktioniert.

(b) Erweitern Sie Signatur und Implementierung von `Bignat` um eine Prozedur

```
fromString :  string -> bignat option
```

die einen String in eine große Zahl umwandelt. Für einen String, der keine natürliche Zahl darstellt, soll die Prozedur `NONE` liefern.

Aufgabe 11.2∗: Zahlensysteme (0+8 Punkte)

Wir können Zahlen zu einer beliebigen Basis B ($2 \leq B \leq 36$) mithilfe von Strings darstellen. Die Ziffern jenseits der 9 werden dabei durch die Buchstaben `A`–`Z` repräsentiert. Der String `"1110"` stellt beispielsweise die Zahl 14 im Binärsystem (Basis 2) dar, und `"7CF"` die Zahl 1999 im Hexadezimalsystem (Basis 16).

Schreiben Sie zwei Prozeduren

(a) `intToStringBase : int -> int -> string`

(b) `stringToIntBase : int -> string -> int option`

die Zahlen zwischen der Dezimaldarstellung und einer Darstellungen zu einer frei wählbaren Basis B ($2 \leq B \leq 36$) konvertieren. Die gewünschte Basis wird als erstes Argument übergeben:

```
intToStringBase 16 1999   =   "7CF"
stringToIntBase 20 "HI"   =   SOME 358
```

Ihre Prozeduren sollen auch negative Zahlen handhaben können, als Vorzeichen in der Stringdarstellung wird `~` verwendet. Enthält der Argumentstring illegale Zeichen oder Ziffern außerhalb des gültigen Bereichs, so liefert `stringToIntBase` den Wert `NONE`. Hinweis: Die Standardstruktur `Char` ist hilfreich.

Aufgabe 11.3: Bäume (6 Punkte)

Ein Typ zur Repräsentation allgemeiner Bäume sei wie folgt gegeben:

```
datatype 'a tree = Tree of 'a * 'a tree list
```

Schreiben Sie Prozeduren, die die folgenden Funktionen auf diesem Typ realisieren:

(a) `size:     'a tree -> int       (* Groesse *)`

(b) `frontier: 'a tree -> 'a list   (* Grenze  *)`

(c) `pre:      'a tree -> 'a list   (* Praefixprojektion *)`

Aufgabe 11.4: Mengen (5 Punkte)

Schreiben Sie ein Modul `Set`, das endliche Mengen über Typen mit Gleichheit mit der folgenden Signatur implementiert:

```
type ''a set

val empty :      ''a set                        (*   {}    *)
val singleton :  ''a -> ''a set                 (*   {a}   *)
val add :        ''a set * ''a     -> ''a set   (* X ∪ {a} *)
val union :      ''a set * ''a set -> ''a set   (*  X ∪ Y  *)
val diff :       ''a set * ''a set -> ''a set   (*  X \ Y  *)

val toList :     ''a set -> ''a list
```

Verwenden Sie dabei

```
type ''a set = ''a list
```

und sorgen Sie dafür, dass in den die Mengen repräsentierenden Listen keine Elemente mehrfach auftreten.

Warum ist es nicht sinnvoll, `''a set` als Typ mit Gleichheit zu definieren?

Aufgabe 11.5: Freie Identifikatoren (8 Punkte)

Das Modul `Phrase` definiert eine abstrakte Syntax für MML (erweitert um `let` und `val`):

```
type constant   = int
type identifier = string

datatype comparison = Equal | Less | LessEq

datatype expression =
        CExp of constant
      | IExp of identifier
      | Appl of identifier * expression list
      | Cond of condition * expression * expression
      | Let  of declaration list * expression

and condition = Comp of expression * comparison * expression

and declaration =
        Val of identifier * expression
      | Fun of identifier * identifier list * expression
```

Schreiben Sie ein Modul `FreeIds` mit der Signatur:

```
val freeIds: Phrase.expression -> Phrase.identifier Set.set
```

so dass `FreeIds.freeIds` die Menge der in einem Ausdruck frei auftretenden Identifikatoren ermittelt. Verwenden Sie dabei einen Satz wechselseitig rekursiver Hilfsprozeduren:

```
expression :    expression      -> identifier set
expressions :   expression list  -> identifier set
condition :     condition       -> identifier set
declaration :   declaration     -> identifier set * identifier set
declarations :  declaration list -> identifier set * identifier set
```

Die Prozeduren liefern jeweils die freien Identifikatoren eines Teilausdrucks, die letzten beiden Prozeduren liefern jeweils ein Paar `(free,bound)`, das neben den freien Identifikatoren auch die durch die Deklarationen gebundenen Identifikatoren enthält.

Im Web finden Sie eine Datei `samples.sml`, die einige Beispielausdrücke enthält, mit denen Sie Ihr Modul bequem testen können.