6. Übungsblatt „Programmiersprachen“ WS 97/98

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 4. Dezember 1997 **Abgabe:** 15. Dezember 1997

Aufgabe VI.1: Parsen von Infix-Ausdrücken ($4 + 9$ Punkte)

Betrachten Sie folgende Repräsentation von Infix-Operatoren:

```
datatype fixity = L | R
type operator = string * int * fixity
val maxPrec = 10000
```

Operatoren werden durch Tripel beschrieben. Die erste Komponente ist eine Zeichenkette, die allein zur Identifikation des Operators dient. Die zweite Komponente ist die Präzedenz, die eine ganze Zahl zwischen 0 einschließlich und `maxPrec` ausschließlich sein muß. Höhere Präzedenzen bedeuten stärkere Bindung. Die dritte Komponente gibt an, ob der Operator links- oder rechtsassoziativ ist. Hier sind einige Beispiele:

```
val plus = ("+", 1, L)
val mal  = ("*", 2, L)
val hoch = ("^", 3, R)
```

Mithilfe dieser Definitionen stellen wir Operatorbäume wie folgt dar:

```
datatype tree =
    Leaf of int
  | Node of operator * tree * tree
```

a. Schreiben Sie eine Prozedur `checkTree` vom Typ `tree -> bool`, die überprüft, ob ein Operatorbaum mit den Präzedenzen und Assoziativitäten seiner Operatoren konsistent ist.

b. Gegeben seien folgende Definitionen:

```
datatype token =
    C of int
  | Op of operator

exception ParseError
```

Schreiben Sie eine Prozedur `parse` vom Typ `token list -> tree`, die die syntaktische Analyse einer Folge von Konstanten und Operatoren unter Berücksichtigung der Operatorpräzedenzen und -assoziativitäten vornimmt und einen Operatorbaum aufbaut. In dem Fall, daß die Folge keinen gültigen Ausdruck darstellt, soll die Ausnahme `ParseError` geworfen werden.

Aufgabe VI.2: Typinferenz für Ausdrücke ($3 + 5 + 6 + 6$ Punkte)

Wir stellen die Ausdrücke einer einfachen funktionalen Sprache, bestehend aus Variablen, einstelligen Lambda-Abstraktionen und Applikationen durch folgenden Datentyp dar:

```
datatype expr =
    X of int              (* Variable *)
  | Abs of int * expr     (* Abstraktion *)
  | App of expr * expr    (* Applikation *)
```

Die Variablen werden einfach durchnumeriert und nur durch ihre Zahl dargestellt.

Unser längerfristiges Ziel ist es, die Typen von Ausdrücken dieser Sprache zu berechnen. Dazu benötigen wir zunächst einen Datentyp, um Typausdrücke zu repräsentieren:

```
datatype typ =
    T of int              (* Typvariable *)
  | Fun of typ * typ      (* Pfeiltyp *)
```

Auch die Typvariablen werden durchnumeriert. Wir gehen davon aus, daß jedes Vorkommen einer Variablen mit Index i in einem Ausdruck denselben Typ besitzt, und zwar den, der durch die Typvariable mit Index i bezeichnet wird.

Um Typen inferieren zu können, brauchen wir noch Substitutionen, also Abbildungen von Typvariablen(-indizes) auf Typen, wie sie in der Vorlesung dargestellt wurden:

```
type subst = (int * typ) list
```

Für Substitutionen muß immer gelten, daß keine der Variablen, die auf den linken Seiten vorkommen, auf einer rechten Seite vorkommt (mit anderen Worten: Substitutionen müssen idempotent sein).

a. Schreiben Sie eine Funktion `applySubst` vom Typ `subst * typ -> typ`, die eine Substitution auf einen Typ anwendet.

b. Implementieren Sie `extendSubst` vom Typ `int * typ * subst -> subst`, die zu gegebenen i, τ und σ die erweiterte Substitution $\sigma[i \leftarrow \tau]$ zurückgibt. Vorher muß allerdings der sogenannte *occur check* durchgeführt werden – das bedeutet, daß überprüft wird, daß die Typvariable mit Nummer i nicht in τ vorkommt. Sollte dies jedoch der Fall sein, so soll die folgende Ausnahme geworfen werden:

   ```
   exception TypeError
   ```

   Weiterhin muß gewährleistet werden, daß die resultierende Substitution die oben angegebene Bedingung der Idempotenz erfüllt.

c. Nun sind wir bereit für die Funktion `unifyTypes` vom Typ `typ * typ * subst -> subst`, die zwei Typen und eine Substitution erhält. Die beiden Typen werden gleichgesetzt (oder *unifiziert*), wobei eventuell neue Erkenntnisse über Typvariablen gewonnen werden. Die Substitution wird um diese neuen Fakten erweitert und zurückgegeben. Implementieren Sie `unifyTypes`.

d. Jetzt soll die Funktion `typeOf` vom Typ `expr -> typ` implementiert werden. Der übergebene Ausdruck wird rekursiv durchlaufen, wobei neue Gleichungen zwischen Typausdrücken erzeugt werden. Das entstehende Gleichungssystem soll mithilfe der oben definierten Funktion `unifyTypes` gelöst werden.

   Hinweis: Da Sie für diese Berechnungen neue Typvariablen werden erzeugen müssen, empfiehlt es sich, zunächst eine Funktion `maxIndex: expr -> int` zu schreiben, die den höchsten in einem Ausdruck verwendeten Variablenindex bestimmt. Die Typvariablen mit kleinerem oder gleichem Index reservieren Sie für die zugehörigen Variablen, alle Typvariablen mit größeren Indizes sind noch frei. Nun können Sie eine Funktion `generate: unit -> typ` schreiben, die bei jedem Aufruf eine neue Typvariable erzeugt, indem intern ein Zähler bei jedem Aufruf hochgezählt wird, der mit dem höchsten Index initialisiert wurde.

   Testen Sie Ihr Programm mindestens mit folgenden Ausdrücken:

```
typeOf (X 1);                                (* T1 *)
typeOf (Abs (1, X 1));                       (* T1->T1 *)
typeOf (Abs (1, Abs (2, App (X 1, X 2))));   (* (T2->T6)->T2->T6 *)
typeOf (App (X 1, X 1));                     (* TypeError *)
```