10. Übungsblatt „Programmiersprachen“ SS 99

Leif Kornstaedt, Prof. Gert Smolka

Ausgabe: 25. Juni 1999 **Abgabe:** 2. Juli 1999

In der 6. Übung haben Sie einen Elaborator für die polymorph typisierte Teilsprache

$$
\begin{aligned}
a \;::=\;& x \\
\mid\;& a_1 \;\; a_2 \\
\mid\;& \texttt{fn}\ x\ \texttt{=>}\ a \\
\mid\;& \texttt{val}\ x\ \texttt{=}\ a_1\ \texttt{in}\ a_2
\end{aligned}
$$

von F2 in SML programmiert. Diesmal sollen Sie den Elaborator in Java schreiben. Sie lernen dabei wichtige objektorientierte Programmiertechniken und können diese im Rahmen eines nichttrivialen Beispiels mit den in SML verwendeten funktionalen Techniken vergleichen.

Lesen Sie zuerst Übungsblatt 6, um sich mit der Aufgabenstellung wieder vertraut zu machen.

Aufgabe X.1: Symboltabellen (6 Punkte)

Implementieren Sie Symboltabellen analog zu Aufgabe VI.1 wie folgt:

```
class Symboltable {
    class NoSuchIdException extends RuntimeException;

    Symboltable();
    void push(String id, Object entry);
    void pop(String id);      // NoSuchIdException
    Object top(String id);    // NoSuchIdException
}
```

Verwenden Sie dabei `java.util.Hashtable` und `java.util.Stack`.

Die Ausnahme `NoSuchIdException` soll eine textuelle Fehlermeldung enthalten, die über die Methode `getMessage` von `Throwable` erfragt werden kann.

Aufgabe X.2: Abstrakte Syntax und Elaboration (12 Punkte)

Implementieren Sie für die Teilsprache eine abstrakte Syntax und einen Elaborator wie folgt:

```
abstract class Expression {
    static Expression var(String x);
    static Expression app(Expression e1, Expression e2);
    static Expression fn(String x, Expression e);
    static Expression val(String x, Expression e1, Expression e2);

    abstract Type elab(Symboltable env);
}
```

Die Typbasis `Type` ist im Web vorgegeben und hat folgende Schnittstelle:

```
abstract class Type {
    class UnificationException extends RuntimeException;

    static Type var();
    static Type cons(String tycon, Type[] tys);
    static void unify(Type ty1, Type ty2);   // UnificationException
    static Type instantiate(Type ty);
    static void beginScheme();
    static void endScheme(Type ty);

    String toString();
}
```

Aufgabe X.3: Interaktiver Interpreter (3 + 7 Punkte)

In dieser Aufgabe soll ein interaktiver Interpreter geschrieben werden.

```
class Interpreter {
    static void initializeTypeEnv(Symboltable typeEnv);
    public static void main(String[] args);
}
```

a. Die Methode `initializeTypeEnv` legt in der übergebenen Symboltabelle die initiale Typumgebung ab:

Bezeichner	Typ
`one`	`int`
`true`	`bool`
`false`	`bool`

b. Im Web ist ein Parser für die Sprache aus Aufgabe X.2 vorgegeben mit folgender Schnittstelle:

```
class Parser {
    class ParserException extends RuntimeException;

    Parser(java.io.InputStream s);
    Expression parseExpression();   // ParserException
}
```

Die Methode `parseExpression` liest einen durch Semikolon beendeten Ausdruck von dem Eingabestrom und liefert seine abstrakte Darstellung. Bei einem Syntaxfehler wirft sie eine Ausnahme. Eine Zeichenkette, die den Fehler beschreibt, kann aus dieser durch `getMessage` extrahiert werden.

Die Methode `main` von `Interpreter` soll eine interaktive Oberfläche realisieren, die in einer Schleife folgende Aktionen durchführt:

- Ausgabe einer Eingabeaufforderung (beispielsweise `"> "`).
- Einlesen eines Audrucks von der Standardeingabe `System.in`.
- Elaborieren des Audrucks in der initialen Umgebung.
- Ausgabe des hergeleiteten Typs oder einer Fehlermeldung.

Eine durch `java Interpreter` eingeleitete Sitzung könnte wie folgt aussehen:

```
> one;
int
> fn x => x;
'a -> 'a
> z;
undeclared identifier z
> fn x => x x;
occur check failed
```

Aufgabe X.4: Symboltabellen mit Klonen (5 Punkte)

Implementieren Sie zu der Klasse `Symboltable` aus Aufgabe X.1 die Methode `clone`, die eine Kopie der Symboltabelle erzeugt. In der Kopie sollen verdeckte Bindungen nicht enthalten sein.

Aufgabe X.5*: Evaluator (5 + 1 + 6 + 3 Punkte)

Implementieren Sie einen Evaluator für obige Teilsprache von F2.

a. Stellen Sie eine Repräsentation von Werten bereit:

$$
\begin{array}{lllll}
w & \in & \text{Wert} & = & \mathbb{Z} \cup \mathbb{B} \cup \text{Pro} \\
 & & \mathbb{Z} & = & \{1\} \\
 & & \mathbb{B} & = & \{\texttt{true}, \texttt{false}\} \\
 & & \text{Pro} & = & \text{Bez} \times \text{Aus} \times \text{WU}
\end{array}
$$

Implementieren Sie außerdem eine Methode, die einen Wert als Zeichenkette darstellt.

b. Schreiben Sie eine Methode

```
static void initializeValueEnv(Symboltable valueEnv);
```

die in der übergebenen Symboltabelle die initiale Wertumgebung ablegt:

Bezeichner	Wert
`one`	`1`
`true`	`true`
`false`	`false`

c. Implementieren sie einen Evaluator, der einen Ausdruck in einer gegebenen Wertumgebung auswertet. Sie müssen hierfür die Klasse `Expression` entsprechend erweitern.

d. Erweitern Sie die interaktive Oberfläche aus Aufgabe X.3b so, dass jeder eingegebene Ausdruck evaluiert wird und das Ergebnis mit dem inferierten Typ ausgegeben wird.

   Eine Sitzung könnte dann wie folgt aussehen:

```
> one;
1: int
> fn x => x;
fn: 'a -> 'a
> z;
undeclared identifier z
> fn x => x x;
occur check failed
```