# Klausur „Programmiersprachen“ SS 99

Leif Kornstaedt, Prof. Gert Smolka
15. Juli 1999

Name: .....
Matrikelnummer: .....
Hauptfach: .....
Fachsemester: .....

## Hinweise

- Es sind keine Unterlagen oder Hilfsmittel erlaubt.
- Die Lösungen müssen direkt ins Klausurheft geschrieben werden.
- Benutzen Sie nur das von uns ausgegebene Notizpapier.
- Die Bearbeitungszeit beträgt **150 Minuten**.
- Die maximal erreichbare Punktzahl ist 194. Mit mindestens 97 erreichten Punkten gilt die Klausur auf jeden Fall als bestanden.
- Täuschungsversuche führen zum sofortigen Ausschluß von der Klausur. In diesem Fall wird die Klausur als *nicht bestanden* gewertet.
- Sie dürfen die Klausur auch mit Bleistift bearbeiten.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | Σ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |
| 6 | 12 | 28 | 20 | 24 | 32 | 16 | 16 | 12 | 18 | 10 | 194 |

**1:** Polymorphe Typen (2 + 2 + 2 Punkte)

Geben Sie die allgemeinsten Typen für die folgenden Prozeduren an.

```
fun f NONE = nil
  | f (SOME x) = [x]
```

```
fun g x y = (f x, f y)
```

```
fun h (x, y) = let val v = f NONE
               in (x::v, y::v)
               end
```

**2:** Enumeration mit SML (12 Punkte)

Schreiben Sie eine Prozedur

```
enumerator: 'a list -> unit -> 'a option
```

die zu einer Liste eine Prozedur mit Zustand liefert, die die Elemente der Liste enumeriert. Nachdem alle Elemente der Liste enumeriert wurden, soll `NONE` geliefert werden.

**3:** Zähler (12 + 12 + 4 Punkte)

(a) Deklarieren Sie eine Struktur `Counter` mit der Signatur

```
signature COUNTER =
    sig
        eqtype counter
        val counter: int -> counter
        val tick:    counter -> int
        val reset:   counter -> unit
    end
```

und Operationen wie folgt:

- `counter` erzeugt bei jedem Aufruf einen neuen Zähler, der wie angegeben initialisiert ist.
- `tick` erhöht den Wert des Zählers um eins und liefert den bisherigen Wert.
- `reset` setzt den Zähler auf seinen initialen Wert zurück.

(b) Sei der Typ

```
type counter =
    {tick:     unit -> int,
     reset:    unit -> unit,
     identity: unit ref}
```

gegeben. Deklarieren Sie eine Prozedur

```
counter: int -> counter
```

die objekt-orientierte Zähler erzeugt, deren Funktionalität den Zählern aus (a) entspricht.

(c) Deklarieren Sie eine Prozedur

```
equal: counter * counter -> bool
```

die objekt-orientierte Zähler auf Gleichheit testet.

**Hinweise:**

- Verschiedene Zähler können denselben Wert haben.
- Denken Sie daran, dass Verbundtypen oft explizit spezifiziert werden müssen.

**4:** Rekursionsoperator ($12 + 8$ Punkte)

(a) Deklarieren Sie eine Prozedur

```
Rek: (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b
```

mit der monomorphe Deklarationen der Form

**fun** `f x =` ⟨*Ausdruck*⟩

in äquivalente Deklarationen der Form

**val** `f = Rek (`**fn** `f =>` **fn** `x =>` ⟨*Ausdruck*⟩`)`

umgeschrieben werden können.

(b) Sei

**fun** `f x =` ⟨*Ausdruck*⟩

eine polymorphe Prozedurdeklaration. Wie kann man diese Deklaration mithilfe des Rekursionsoperators Rek aus (a) in eine äquivalente Deklaration

**val** `f = ...`

umschreiben?

## **5:** Funktoren (20 + 4 Punkte)

Seien die Signaturen

```
signature KEY_ENTRY =
    sig
        eqtype key
        type entry
    end

signature TABLE =
    sig
        eqtype key
        type entry
        type table
        val  empty:  unit -> table
        val  insert: table * key * entry -> unit
        val  lookup: table * key -> entry option
    end
```

gegeben. Strukturen mit der Signatur `TABLE` sollen Tabellen als imperative Datenstrukturen realisieren.

(a) Schreiben Sie einen Funktor `MakeTable`, der Strukturen, die der Signatur `KEY_ENTRY` genügen, auf Strukturen abbildet, die der Signatur `TABLE` genügen.

(b) Deklarieren Sie mit Hilfe des Funktors `MakeTable` eine Struktur `Table` mit Schlüsseln des Typs `string` und Einträgen des Typs `int`.

**6:** Dynamische Semantik (2 + 4 + 2 + 6 + 6 + 6 + 6 Punkte)

Sie sollen die dynamische Semantik einer SML-artigen Sprache R mit Inferenzregeln spezifizieren. Die abstrakte Syntax von R ist:

| | | | |
|---|---|---|---|
| $x$ | $\in$ | Bez | |
| $a$ | $\in$ | Aus | |
| $a$ | $::=$ | `()` | Unit |
| | $\mid$ | $x$ | Bezeichner |
| | $\mid$ | **`fn`** $x$ **`=>`** $a$ | Abstraktion |
| | $\mid$ | $a_1\ a_2$ | Applikation |
| | $\mid$ | **`ref`** $a$ | Referenzerzeugung |
| | $\mid$ | `!`$a$ | Dereferenzierung |
| | $\mid$ | $a_1$ **`:=`** $a_2$ | Zuweisung |

Die dynamische Semantik verwendet Werte, Umgebungen und Zustände wie folgt:

| | | | | | |
|---|---|---|---|---|---|
| $w$ | $\in$ | Wert | $=$ | $\{\emptyset\} \cup \text{Ref} \cup \text{Pro}$ | Wert |
| $r$ | $\in$ | Ref | | | Referenz |
| | | Pro | $=$ | $\text{Bez} \times \text{Aus} \times \text{WU}$ | Prozedur |
| $W$ | $\in$ | WU | $=$ | $\text{Bez} \xrightarrow{\text{fin}} \text{Wert}$ | Wertumgebung |
| $\sigma$ | $\in$ | Zust | $=$ | $\text{Ref} \xrightarrow{\text{fin}} \text{Wert}$ | Zustand |

Dabei ist eine Funktion

$$\text{Next} : \text{Fin}(\text{Ref}) \to \text{Ref}$$

mit

$$\forall R \in \text{Fin}(\text{Ref}) : \text{Next}(R) \notin R$$

für die Erzeugung von Referenzen gegeben. Fin(Ref) ist die Menge aller endlichen Teilmengen von Ref.

Die dynamische Semantik wird mit Transitionen der Form

$$\sigma, W \vdash a \Rightarrow w, \sigma'$$

formuliert. Ergänzen Sie die Prämissen der nachfolgenden Inferenzregeln so, dass diese eine SML-artige dynamische Semantik für R spezifizieren.

$$\frac{}{\sigma, W \vdash \texttt{()} \Rightarrow \emptyset, \sigma}$$

$$\frac{}{\sigma, W \vdash x \Rightarrow w, \sigma}$$

$$\frac{}{\sigma, W \vdash \texttt{fn}\ \ x\ \ \texttt{=>}\ \ a \Rightarrow (x, a, W), \sigma}$$

$$\frac{}{\sigma, W \vdash a_1\ a_2 \Rightarrow w', \sigma'''}$$

$$\frac{}{\sigma, W \vdash \texttt{ref}\ \ a \Rightarrow r, \sigma''}$$

$$\frac{}{\sigma, W \vdash\ !\, a \Rightarrow w, \sigma'}$$

$$\frac{}{\sigma, W \vdash a_1\ \texttt{:=}\ a_2 \Rightarrow \emptyset, \sigma'''}$$

**7:** Enumeration mit Java (16 Punkte)

Vervollständigen Sie die folgenden Deklarationen um die Implementation der dynamischen Methode `enumerator`, die einen Enumerator für `this` liefern soll. Nachdem alle Elemente der Liste enumeriert wurden, soll `null` geliefert werden.

```
interface Enumerator {
    Object nextElement();
}

class List {
    final Object head;
    final List tail;
    List(Object o, List l) { head = o; tail = l; }
    Enumerator enumerator() {









    }
}
```

## 8: Ausdrücke in Java (16 Punkte)

Vervollständigen Sie die nachfolgende Klassendeklaration so, dass damit arithmetische Ausdrücke mit Produkten erzeugt, zerlegt und evaluiert werden können.

```
abstract class Expression {
    abstract int eval();

    static class Con extends Expression {


        Con(int x) {


        }









    }
    static class Prod extends Expression {




        Prod(Expression x, Expression y) {




        }









    }
}
```

**9:** Casts (12 Punkte)

Seien die folgenden Typdeklarationen gegeben:

```
interface A {}
class B implements A {}
class C implements A {}
class D extends B {}

class Test {
    public static void main(String[] args) {
        B b  = new B();
        A a  = (A) b;
        B b1 = (B) a;
        C c  = (C) b;
        C c1 = (C) a;
        D d  = (D) b;
        D d1 = (D) a;
    }
}
```

Tragen Sie in der folgenden Tabelle für jeden der obigen Casts ein, ob er gemäß der

(a) statischen Semantik überflüssig, notwendig oder illegal ist (exklusives Oder). Dabei sollen überflüssige und notwendige Casts natürlich legal sein.

(b) dynamischen Semantik scheitert oder erfolgreich ist.

| Cast | statisch überflüssig/notwendig/illegal | dynamisch erfolgreich/scheitert |
|---|---|---|
| `A a  = (A) b;` | | |
| `B b1 = (B) a;` | | |
| `C c  = (C) b;` | | |
| `C c1 = (C) a;` | | |
| `D d  = (D) b;` | | |
| `D d1 = (D) a;` | | |

**10:** Konstruktoren $(4 + 6 + 4 + 4$ Punkte)

Seien die folgenden Klassendeklarationen gegeben:

```
class A {
    A() {
        println("A");
    }
    void println(String s) {
        System.out.println(s);
    }
}

class B extends B {
}

class C extends B {
    C() {
        println("C");
    }
    public static void main(String[] args) {
        new C();
    }
}
```

(a) Was wird bei der Ausführung von `main` ausgegeben?

(b) Erweitern sie die obigen Klassendeklarationen so, dass implizite Konstruktoraufrufe durch explizite Konstruktoraufrufe ersetzt werden. (Schreiben Sie die Erweiterungen direkt in die obigen Deklarationen.)

(c) Warum sind die folgenden Klassendeklarationen statisch unzulässig?

```
class A {
    A(String s) {
        println(s);
    }
    void println(String s) {
        System.out.println(s);
    }
}

class B extends A {
}

class C extends B {
    C() {
        println("C");
    }
    public static void main(String[] args) {
        new C();
    }
}
```

(d) Erweitern sie die Klassendeklarationen in (c) durch eine Konstruktordeklaration für `B` so, dass die Deklarationen statisch zulässig werden. (Schreiben Sie die Erweiterungen direkt in die obigen Deklarationen.)

**11:** Erweiterung, Verdeckung, Überladung, Überschreibung (10 Punkte)

Seien folgende Klassendeklarationen gegeben:

```
class A {
    int a = 1;
    void v() {}
    static void u() {}
    void w() {}
}

class B extends A {

    int a = 2;

    int b = 3;

    void v() {}

    static void u() {}

    void w(int x) {}

}
```

Geben Sie für jede Deklaration von `B` an, ob es sich um eine Erweiterung, Verdeckung, Überladung oder Überschreibung handelt (exklusives Oder).