EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Beispiel-Klausuraufgaben
zur Vorlesung **Programmiersprachen 1**
im Sommersemester 2025

Matrikelnummer: ☐

Name: ☐

Diese Beispielklausur besteht aus 2 Programmieraufgaben und vier Multiple-Choice-Fragen. Jede Textaufgabe wird mit 8 Punkten bewertet; jede Multiple-Choice Frage wird mit 2 Punkten bewertet.

Die Klausur besteht aus 2 Programmieraufgaben und 30 Multiple-Choice-Fragen. Jede Textaufgabe wird mit 15 Punkten bewertet und jede Multiple-Choice Frage wird mit 2 Punkten bewertet. Insgesamt sind also 90 Punkte zu erreichen.

Jede der vier Fragen hat genau eine richtige Antwort. Für jede Frage können Sie keine Antwort, eine Antwort, oder mehrere Antworten geben indem Sie die entsprechenden Felder ankreuzen. Das bedeutet:

- Wenn Sie die Antwort sicher wissen, kreuzen Sie nur diese an.
- Wenn Sie sich nicht sicher sind aber einige Antworten ausschließen können, kreuzen Sie die Antworten an, die korrekt sein könnten.
- Wenn Sie überhaupt nicht wissen, was die Antwort sein könnte und nichts ausschließen können, kreuzen Sie nichts an.

Die Antworten werden so bewertet, dass zufälliges Raten null Punkte ergibt, also eine falsche Antwort Minuspunkte gibt.

Wir benutzen die Konvention dass ☒ ”Antwort ausgewählt” bedeutet, und sowohl ☐ als auch ■ bedeuten ”Antwort *nicht* gewählt”.

Programmieraufgabe 1

Vervollständigen Sie den `Letcc` Fall im Interpreter aus der Vorlesung über First-Class Continuations:

```
def eval(e: Exp, env: Env, k: Value => Nothing) : Nothing = e match {
  ...
  case Letcc(param, body) =>
```

Programmieraufgabe 2

Das folgende Scala Programm berechnet das kartesische Produkt der Listen `l1` and `l2`, d.h., eine Liste von Paaren `(x, y)`, mit `x` aus Liste `l1` and `y` aus Liste `l2`:

```
def map(xs: List[Int])(f: Int => (Int, Int)): List[(Int, Int)] =
  xs match {
    case Nil     => Nil
    case x :: xs => f(x) :: map(xs)(f)
  }

def flatMap(xs: List[Int])(f: Int => List[(Int, Int)]): List[(Int, Int)] =
  xs match {
    case Nil     => Nil
    case x :: xs => f(x) ++ flatMap(xs)(f)
  }

def product(l1: List[Int], l2: List[Int]) =
  flatMap(l1)(x =>
    map(l2)(y =>
      (x, y)))
```

a. Lambda-Liften Sie dieses Programm.

b. Defunktionalisieren Sie das Programm aus Teilaufgabe a.

Frage 1

Was ist ein Tail-Call?

Ein Tail-Call ist . . .

- *a* ☐ . . . der letzte Funktionsaufruf im Quellcode einer Funktion.
- *b* ☐ . . . der möglicherweise zuletzt ausgeführte Funktionsaufruf in einer Funktion zur Laufzeit.
- *c* ☐ . . . der möglicherweise zuletzt ausgeführte Funktionsaufruf in einem Programm zur Laufzeit.
- *d* ☐ . . . der Aufruf einer Continuation.

Frage 2

Was sind Monadentransformer?

- *a* ☐ Ein Monadentransformer kann häufig verwendet werden, um Monaden miteinander zu kombinieren.
- *b* ☐ Ein Monadentransformer transformiert die Reader-Monade in die State-Monade.
- *c* ☐ Ein Monadentransformer transformiert ein Programm in den monadischen Stil.
- *d* ☐ Ein Monadentransformer macht jeden Typkonstruktor zu einer Monade.

Frage 3

Welche der folgenden Aussagen über Auswertungsstrategien ist korrekt im ungetypten λ-Kalkül (= FAE)?

- *a* ☐ Unter Call-by-Value wird ein Funktionsargument nach der Auswertung gecached.
- *b* ☐ Jedes Programm, welches unter Call-by-Name terminiert, terminiert auch unter Call-by-Value.
- *c* ☐ Jedes Programm, welches unter Call-by-Value terminiert, terminiert auch unter Call-by-Need.
- *d* ☐ Unter Call-by-Name wird jedes Funktionsargument höchstens einmal ausgewertet.

Frage 4

Was könnte das Ergebnis des Aufrufs `eval(NewBox(42), Map(), Map())` im Interpreter für BCFAE (FAE mit Zustand) aus der Vorlesung sein?

Das Ergebnis könnte . . .

- *a* ☐ . . . `(AddressV("x"), Map("x" -> NumV(42)))` sein.
- *b* ☐ . . . `(AddressV(1), Map(1 -> NumV(42)))` sein.
- *c* ☐ . . . `(AddressV(42), Map())` sein.
- *d* ☐ . . . `(AddressV(1), Map(1 -> 42))` sein.