Name, Vorname: Matrikelnummer:

Sprachkonzepte – Probeklausur

Bearbeitungszeit 90 Minuten. Es sind keine Hilfsmittel zugelassen.

Aufgabe 1: Sprachen und reguläre Ausdrücke *(14 Punkte)*

a) Nennen Sie die drei in der Lehrveranstaltung besprochenen Aspekte einer Programmiersprache, jeweils mit ihrer Unterteilung in zwei Teilaspekte. *(3 Punkte)*

b) Welcher der Teilaspekte aus a) wird üblicherweise mit regulären Ausdrücken definiert? Nennen Sie vier konkrete Beispiele, die in gängigen Programmiersprachen unter diesen Teilaspekt fallen. *(5 Punkte)*

c) Viele Programmiersprachen, insbesondere auch Scriptsprachen, bieten in ihrer Bibliothek oder mitunter sogar auf Sprachebene ausgefeilte Implementierungen von regulären Ausdrücken. Warum ist das so? Geben Sie zwei Beispiel wofür man die regulären Ausdrücke nutzt. (3 Punkte)

d) Erklären Sie den folgenden in der Syntax von ANTLR4 geschriebenen regulären Ausdruck. Gehen Sie dabei auch darauf ein, warum die Verwendung von **`.*`** statt **`.*?`** falsch wäre: *(3 Punkte)*

```
'/*' .*? '*/'
```

Aufgabe 2: Sprachen und Grammatiken *(21 Punkte)*

Betrachten Sie die folgende ANTLR4 Grammatik für Unified Resource Identifiers (stark vereinfacht):

```
// Uri.g4
grammar Uri;
// Parser Rules:
uri : url EOF ;
url : scheme ':' '//'? authority ('/' path?)? ('?' query)? ('#' frag)? ;
scheme : string ;
authority : (login '@')? host (':' port)? ;
login : user (':' password)? ;
user : string ;
password : string ;
host : string ;
port : DIGITS ;
path : string ('/' string)* '/'?  ;
query : searchparameter ('&' searchparameter)* ;
searchparameter : string ('=' string)?  ;
frag : string ;
string : STRING | DIGITS ;
// Lexer Rules:
DIGITS : [0-9]+ ;
STRING : [a-zA-Z~0-9] [a-zA-Z0-9.+-]* ;
```

a) Zeichnen Sie die Ableitungsbäume (Parse Trees) für die folgenden beiden URIs: *(15 Punkte)*

```
https://www.htwg-konstanz.de/~drachen/
mailto:drachenfels@htwg-konstanz.de?subject=Klausur
```

b) Die obige Grammatik akzeptiert auch falsche URIs, etwa die beiden aus a) mit verwechseltem Schema:

```
mailto://www.htwg-konstanz.de/~drachen/
https:drachenfels@htwg-konstanz.de?subject=Klausur
```

Tatsächlich sind einige der in der Grammatik als optional gekennzeichneten Angaben hinter dem Doppelpunkt je nach Schema verpflichtend bzw. unzulässig.

- Was müsste man an der Grammatik ändern, um das Problem zu lösen?
- Was könnte man bei unveränderter Grammatik tun?
- Wie beurteilen Sie die beiden Lösungswege vor dem Hintergrund, dass es neben **`https`** und **`mailto`** viele weitere Schemas gibt?

(6 Punkte)

Aufgabe 3: Begriffe der funktionalen Programmierung *(10 Punkte)*

In der funktionalen Programmierung ersetzt man Schleifen durch Rekursion und Werte durch Funktionen, die den Wert bei Bedarf berechnen.

a) Welche Einschränkungen gelten für Funktionen im Sinne der funktionalen Programmierung im Vergleich zu Prozeduren der imperativen Programmierung *(in Java als Methoden bezeichnet, in C als Funktionen)*? *(2 Punkte)*

b) Was sind Funktionen höherer Ordnung? *(2 Punkte)*

c) Was sind Lambdas und welche Rolle spielen Sie im Zusammenhang mit Frage b)? *(2 Punkte)*

d) Was sind Closures? *(1 Punkt)*

e) Wie funktioniert Currying? *(3 Punkte)*
Hinweis: Currying ist Grundlage dafür, dass rein funktionale Programmiersprachen theoretisch keine Funktionen mit mehr als einem Parameter unterstützen müssen.

Aufgabe 4: Anwendung der funktionalen Programmierung *(14 Punkte)*

a) Was lässt sich über Laufzeit und Speicherbedarf funktionaler Programme im Vergleich zu imperativen Programmen sagen, wenn man einen nicht optimierenden Compiler annimmt? *(4 Punkte)*
Hinweis: beachten Sie den einführenden Satz bei Aufgabe 3

b) Betrachten Sie die folgenden beiden rekursiven **summe**-Funktionen, die beide das gleiche berechnen.
- Welche davon lässt sich leicht optimieren, indem man die Rekursion in eine Schleife umwandelt?
- Wie nennt man die darin verwendete optimierungsfreundliche Rekursionsform?
- Geben Sie eine mittels Schleife optimierte Implementierung der betreffenden Funktion an, die so auch ein optimierender Compiler aus der ursprünglichen Funktion ableiten könnte.

(10 Punkte)

```
public static void main(String[] args) {
   int s;
   s = summe(11, 13);
   s = summe(11, 13, 0);
}

private static int summe(int von, int bis) {
    if (von >= bis) {
        return von;
    }
    return von + summe(von + 1, bis);
}

private static int summe(int von, int bis, int s) {
    if (von >= bis) {
        return von + s;
    }
    return summe(von + 1, bis, von + s);
}
```

Aufgabe 5: Logische Programmierung *(20 Punkte)*

a) Betrachten Sie die folgenden Prolog-Fakten mit Klausurnoten:

```
note(berta,1.3).
note(bruno,2.0).
note(erna,2.0).
```

Was liefert der Prolog-Interpreter bei den folgenden Anfragen jeweils als erste Antwort?
Bei welchen Anfragen gibt es weitere Antworten und welche Antworten sind das? *(2 Punkte)*

```
?- note(berta,1.3)
?- note(bruno,N)
?- note(X,2.0)
```

b) Formulieren Sie eine Regel **`bestanden(Person)`**, die prüft, ob eine Person die Klausur bestanden hat. Die Regel soll erfolgreich sein, wenn ein **`note`**-Fakt für die Person existiert und die Note darin kleiner oder gleich **`4.0`** ist. *(3 Punkte)*

c) Welche erste Antwort und welche Folgeantworten liefern jeweils die folgenden Anfragen mit Ihrer Regel aus b) und den Fakten aus a)? *(2 Punkte)*

```
?- bestanden(berta)
?- bestanden(X)
?- bestanden(willi)
```

d) Gegeben ist folgende Regel, die die Durchschnittsnote einer Liste von Noten berechnet:

```
durchschnitt(Liste, D) :- summe(Liste,S), anzahl(Liste,N), D is S / N.
```

Bei einem Aufruf **`durchschnitt([1.3,2.0,2.0], D)`** wird **`D`** mit dem berechnet Durchschnitt unifiziert. Ergänzen Sie die erforderlichen **`summe`**- und **`anzahl`**-Regeln. *(10 Punkte)*

e) Mit welchen Werten werden im folgenden Ausdruck die Variablen unifiziert? *(3 Punkte)*

```
[[berta,A],[B,2.0],[erna,2.0]|C] = [[X,1.3],[bruno,2.0],Y,[willi,5.0],[B,A]]
```

Aufgabe 6: Typinferenz *(6 Punkte)*

a) Was ist Typinferenz und wofür wird sie verwendet? *(3 Punkte)*

b) An welchen drei Stellen in den folgenden Java-Anweisungen verwendet der Compiler Typinferenz und zu jeweils welchem Zweck? *(3 Punkte)*

```
double n = args.length > 0 ? 1 : 2.0;
var list = List.of(args);
for (var s : list) { }
```

Hinweis: die verwendete Bibliotheksmethode ist wie folgt definiert:

```
public interface List<E> ... {
    ...
    static <E> List<E> of(E... elements)
    ...
}
```

Aufgabe 7: Parametrische Polymorphie *(7 Punkte)*

a) Was ist parametrische Polymorphie und was ist ihr Nutzen? *(2 Punkte)*

b) Wie implementiert Java parametrische Polymorphie und was bedeutet das konkret für die übersetzte Version von **`java.util.List`**? *(2 Punkte)*

```
// Ausschnitt aus List.java:
public interface List<E> ... {
    ...
    boolean add(E e);
    E get(int index);
    ...
}
```

c) In den folgenden Java-Anweisungen finden einige Typanpassungen statt.
Schreiben Sie die Anweisungen neu mit überall explizit ausgeschriebenen Typecast-Operatoren.
Berücksichtigen Sie dabei das in b) Gesagte. *(3 Punkte)*

```
List<String> list = new ArrayList<String>();
list.add("Aufgabe 6");
String x = list.get(0);
```

Aufgabe 8: Namen, Bindungen, Scopes *(8 Punkte)*

a) python3 verwendet für Variablen statisches Scoping, während sh dynamisches Scoping verwendet. Geben Sie für beide Programm an, was sie auf den Bildschirm schreiben. *(4 Punkte)*

```
#!/usr/bin/python3
def f():
    global s
    s = 'Wert aus f'

def g():
    s = 'Wert aus g'
    f()

s = 'Wert aus main'
f()
print(s, end='\n')
s = 'Wert aus main'
g()
print(s, end='\n')
```

```
#!/bin/sh
function f()
{
    s='Wert aus f'
}

function g()
{
    local s='Wert aus g'
    f
}

s='Wert aus main'
f
echo $s
s='Wert aus main'
g
echo -n $s
```

b) Blöcke wie der Rumpf einer Funktion bilden in allen gängigen Programmiersprachen einen Scope. Aber je nach Sprache gilt die Bindung eines Variablennamens nicht ab Beginn des Scopes. Erklären Sie warum das folgende C-Programm 1 und 2 ausgibt, das python-Programm aber mit einer Fehlermeldung abbricht. *(4 Punkte)*

```
#include <stdio.h>
int n = 1;

void f() {
    printf("%d\n", n);
    int n = 2;
    printf("%d\n", n);
}

int main(void) {
    f();
}
```

```
#!/usr/bin/python3
n = 1

def f():
    print(n)
    n = 2
    print(n)

f()
```