VU Grundlagen der Programmkonstruktion — Übungsteil

Organisatorisches

Wie sollen Ihre Lösungen aussehen

Beachten Sie, dass nicht die konkrete Lösung im Vordergrund steht, sondern der Weg, wie Sie zu dieser gelangen. Geben Sie daher bei Aufgaben immer auch den Rechenweg an. Bei jeder Aufgabe ist ein Beispiel angeführt, wie dieser aussehen soll. Ohne Angabe des Rechenwegs erhalten Sie für die Aufgabe keine Punkte.

Geben Sie ihre Lösungen im TUWEL als (bevorzugt) Text-Dokument (Erweiterung `.txt`) oder als PDF-Dokument (Erweiterung `.pdf`) ab.

Gruppenarbeiten

Sie können die Aufgaben in Gruppenn zu 2-6 Personen lösen. Die Zusammenstellung und Organisation der Gruppen ist dabei Ihnen überlassen.

Wenn Sie die Aufgabe in einer Gruppe gelöst haben, so soll EIN Gruppenmitglied die Aufgabe in TUWEL abgeben. Die Namen MIT Matrikelnummern aller Gruppenmitglieder muss im abgegebenden Dokument am Anfang ihres Dokuments vermerkt sein.

Abgabe Ihrer Lösungen

Die Abgabe Ihrer Lösungen erfolgt elektronisch via TUWEL (TU Wien E-Learning Center). Sie finden einen Link dorthin am Beginn der LVA-Seite im TISS ("Zum TUWEL Online-Kurs").

Die Abgabe muss via TUWEL bis spätestens 27. Oktober 2011, 12:00, erfolgen. Nachträgliche Abgaben werden nicht berücksichtigt.

Fragen

Bei Fragen zu TUWEL wenden Sie sich bitte zunächst an das TUWEL-Hilfe-System und Ihre Kollegen. Sollte in der Angabe etwas unklar sein, sehen Sie bitte im TUWEL nach, ob dort bereits ergänzende Anmerkungen zum Übungsteil stehen. Wichtige Ankündigungen zum Übungsteil werden auch in der Vorlesung und online bekannt gegeben.

2. Übungsaufgabe

1 Bedingungen

Manche if-Anweisungen können in `switch-case`-Anweisung und in Ausdrücke mit dem Bedingungsoperator `?` umgewandelt werden.

Beispiel

```
int i = /* ... */;

boolean istNull;

if(i == 0) {
    istNull = true;
} else {
    istNull = false;
}
```

Diese if-Anweisung entspricht folgender `switch-case`-Anweisung:

```
int i = /* ... */;

boolean istNull;

switch(i) {
    case 0: istNull = true;
            break;
    default: istNull = false;
}
```

Weiters kann die if-Anweisung in einen Ausdruck mit dem Bedingungsoperator `?` umgewandelt werden:

```
int i = /* ... */;

boolean istNull = i == 0 ? true : false;
```

Man kann auch direkt das Ergebnis des Vergleichs `i == 0` in die Variable `istNull` übertragen:

```
int i = /* ... */;

boolean istNull = i == 0;
```

if-Anweisung

Wandeln Sie folgende if-Anweisung in eine äquivalente `switch-case`-Anweisung (vergessen Sie die `break`-Anweisungen nicht), in einen Ausdruck mit dem Bedingungsoperator `?` und in einen Ausdruck mit boolschen Operationen und Vergleichen um:

```
/* Ermitteln, ob ein Wochentag ein Arbeitstag ist oder nicht */

int wochentag = /* ... */;

boolean arbeitstag;

if(wochentag == 6) {
    // Samstag
    arbeitstag = false;
} else if(wochentag == 7) {
    // Sonntag
    arbeitstag = false;
} else {
    arbeitstag = true;
}
```

switch-case-Anweisung

`case`-Sprungmarken sollten immer mit einem `break;` abgeschlossen werden, da es ansonsten zu einem "Fall through" kommt, d.h., es werden auch Anweisungen im folgenden `case`-Teil ausgeführt.

switch-Anweisung 1

Wenn `i == 1` wird im folgenden Programmcode zunächst `i += 2`, danach `i += 3` und abschließend `i += 4` ausgeführt. Wenn `i == 2` wird nur `i += 3` und `i += 4` ausgeführt.

```
/* switch-Anweisung 1 */

int i = /* ... */;

switch(i) {
    case 0: i++;
    case 1: i += 2;
    case 2: i += 3;
    default: i += 4;
}
```

switch-Anweisung 2

In nächsten Code-Fragment wird für `i == 0` zuerst `i++` und danach `i += 2` aufgerufen. Die darauf folgende `break`-Anweisung beendet danach die `switch`-Anweisung:

```
/* switch-Anweisung 2 */

int i = /* ... */;

switch(i) {
    case 0: i++;
    case 1: i += 2; break;
    case 2: i += 3;
    default: i += 4;
}
```

Aufgabenstellung

Wandeln Sie *beide* `switch`-Anweisungen in äquivalente `if`-Ausdrücke um. Sie können dabei auch die Ausdrücke arithmetisch vereinfachen.

Lösung

Beachen Sie, dass es mehr als eine mögliche Lösung gibt.

Aufgabe 1.1 - switch-case

```
boolean arbeitstag;

switch(wochentag) {
case 6 : // Samstag
    arbeitstag = false;
    break;
case 7 : // Sonntag
    arbeitstag = false;
    break;
default:
    arbeitstag = true;
    break;
}
```

Aufgabe 1.1 - Bedingungsoperator

```
boolean arbeitstag = wochentag == 6 || wochentag == 7 ?  false :  true;
```

Aufgabe 1.1 - Boolscher Ausdrucl

```
boolean arbeitstag = ! ( wochentag == 6 || wochentag == 7 );
```

Aufgabe 1.2

Das Ziel dieser Aufgabe war es, zu zeigen, dass einfache `switch-case`-Anweisungen sehr komplex werden können. Sie sollten es vermeiden, solche switch-case-Anweisungen zu verwenden. Um gut programmieren zu können, reicht es erfahrungsgemäß nicht, nur "guten" Programmierstil zu präsentieren. Es ist auch nötig, die Auswirkungen von schlechten Programmierstil zu begreifen und nachzuvollziehen.

Aufgabe 1.2.1

```
int i = /* ... */;

if(i == 0 || i == 1 || i == 2) {
    if(i == 0 || i == 1) {
        if(i == 0) {
            i++;
        }
        i += 2;
    }
    i += 3;
```

```
}
i += 4;
```

Aufgabe 1.2.2

```
int i = /* ... */;
if(i == 0 || i == 1) {
    if(i == 0) {
        i++;
    }
    i += 2;
} else {
    if(i == 2) {
        i += 3;
    }
    i += 4;
}
```

2 Schleifen

`for`-Schleifen, `while`-Schleifen und `do-while`-Schleifen können ineinander umgewandelt werden.

Beispiel

Die folgenden Code-Fragmente berechnen mit unterschiedlichen Schleifentypen die Summen von eins bis n:

```
/* for-Schleife */
int n = /* ... */;

int sum = 0;

for(int i = 1; i <= n; i++) {
    sum += i;
}
```

```
/* while-Schleife */
int n = /* ... */;

int sum = 0;
int i = 1;
```

```
while(i <= n) {
    sum += i;
    i++;
}
```

```
/* do-while-Schleife */
int n = /* ...*/;

int sum = 0;
int i = 1;

do {
    sum += i;
    i++;
} while(i <= n);
```

Beachten Sie, dass die `for`- und `while`-Schleife auch nie ausgeführt werden kann, da die Bedingung am Anfang jedes Schleifendurchlaufs überprüft wird. Im Gegensatz dazu wird die `do-while`-Schleife mindestens einmal durchlaufen und die Bedingung am Ende überprüft. Im Beispiel oben hätten wir somit unterschiedliche Ergebnisse, wenn der Wert von n kleiner als 1 wäre. Diese Fälle können wir mit einer `if`-Anweisung ausschließen:

```
/* do-while-Schleife mit if */
int n = /* ... */;

int sum = 0;
int i = 1;

if(n >= 1) {
    do {
        sum += i;
        i++;
    } while(i <= n);
}
```

Aufgabenstellung

Wandeln Sie die beiden folgenden Schleifen in die jeweils anderen Schleifentypen um. Verwenden Sie (wenn nötig) zusätzlich eine if-Anweisung um die `do-while`-Schleife herum.

```
/* for-Schleife */
int n = /* ... */;
```

```
int fac = 1;

for(int i = 1; i < n; i++) {
    fac *= i;
}
```

```
/* while-Schleife - Heron-Verfahren */
int n = /* ... */;

int i = 0;

double a = 2;
double sqrt_a = 1;

while(i <= n) {
    sqrt_a = (sqrt_a + a / sqrt_a) / 2.;

    i++;
}
```

Lösung

Beachten Sie, dass es keine eindeutige Lösung gibt.

Faktorielle: while-Schleife:

```
int n = /* ... */;

int fac = 1;

int i = 1;
while(i < n) {
    fac *= i;
    i++;
}
```

Faktorielle: do-while-Schleife (die im Fall n ¡= 0 zusätzlich auftretende Mutliplikation mit 1 hat keine Auswirkungen).

```
int n = /* ... */;

int fac = 1;
```

```
int i = 1;
do {
    fac *= i; // Wenn n <= 0 wird 1x zusaetzlich fac *= 1 ausgefuehrt.
    i++;
} while(i < n);
```

Heron-Verfahren - do-while-Schleife:

```
int n = /* ... */;

int i = 0;

double a = 2;
double sqrtA = 1;

if(i <= n) {
    do {
        sqrtA = (sqrtA + a / sqrtA) / 2.;
        i++;
    } while(i <= n);
}
```

3 Algorithmen

Teilen Sie ihre Matrikelnummer (und die aller Ihrer Gruppenmitglieder) in zwei Teile: Die ersten vier Stellen weisen Sie der Variablen m, die letzten drei der Variablen n zu (d.h., sollte Ihre Matrikelnummer 1234567 sein, so ist $m = 1234$ und $n = 567$).

Im Skriptum findet sich in Listing 2.1 der Euklidische Algorithmus. Lassen Sie diesen mit m und n laufen und schreiben Sie die Werte von m und n sowie den Verarbeitungsschritt in eine Tabelle (sollten Sie mehr als 15 Zeilen benötigen, reichen die ersten 15 Zeilen aus). Verwenden Sie dabei die Tabelle 2.4 im Skriptum als Vorbild.

Verletzen Sie die Vorbedingung $m > 0$ und $n > 0$ und erstellen Sie eine Tabelle mit zumindest fünf Zeilen mit den negierten Werten von m und n (d.h., falls Ihre Matrikelnummer 1234567 ist mit $m = -1234$ und $n = -567$).

Beispiel

Siehe Tabelle 2.4 im Skriptum.

Lösung

Für `m = 1234` und `n = 567`:

Anweisung	m	n
	1234	567
`m = m - n`	667	567
`m = m - n`	100	567
`n = n - m`	100	467
`n = n - m`	100	367
`n = n - m`	100	267
`n = n - m`	100	167
`n = n - m`	100	67
`m = m - n`	33	67
`n = n - m`	33	34
`m = m - n`	33	1
...	...	...
`m = m - n`	1	1

Mit negierten Werten:

Anweisung	m	n
	-1234	-567
`n = n - m`	-1234	667
`n = n - m`	-1234	1901
`n = n - m`	-1234	3135
`n = n - m`	-1234	4369
`n = n - m`	-1234	5603

4 Rekursion und Iteration

Die folgenden Funktionen berechnen die Fibonacci-Zahlen rekursiv und iterativ:

```
public static int rfib(int n) {
    if(n == 0) {
        return 0;
    } else if(n == 1) {
        return 1;
    } else {
        return rfib(n - 1) + rfib(n - 2);
    }
}
```

```
public static int ifib(int n) {
    if(n == 0) {
        return 0;
    } else if(n == 1) {
        return 1;
    } else {
        int a = 0;
        int b = 1;

        for(int i = 2; i <= n; i++) {
            b = a + b;
            a = b - a;
        }

        return b;
    }
}
```

Berechnen Sie `rfib(6)` und geben Sie dabei die Reihenfolge der Aufrufe von `rfib` mit Information darüber an, der wievielte Aufruf von `rfib` gereade bearbeitet wird und von welchem Aufruf dieser erfolgte (siehe Beispiel unten). Danach berechnen Sie `ifib(12)` und geben Sie für jeden Schleifendurchlauf die Werte von i, a und b an (jeweils am Ende des Schleifendurchlaufs).

Beispiel

Für `rfib(3)` geben Sie z.B. folgende Tabelle an:

Index	Aufgerufen von Index	Aufruf	Ausgeführte Anweisung
1	-	`rfib(3)`	`rfib(2) + rfib(1)`
2	1	`rfib(2)`	`rfib(1) + rfib(0)`
3	2	`rfib(1)`	`return 1`
4	2	`rfib(0)`	`return 0`
5	1	`rfib(1)`	`return 1`

Für `ifib(3)` lautet die Lösung

Wert in `i`	Wert in `a`	Wert in `b`
-	0	1
2	1	1
3	1	2

Lösung

Index	Aufgerufen von	Aufruf	Anweisung
1	-	`rfib(6)`	`rfib(5) + rfib(4)`
2	1	`rfib(5)`	`rfib(4) + rfib(3)`
3	2	`rfib(4)`	`rfib(3) + rfib(2)`
4	3	`rfib(3)`	`rfib(2) + rfib(1)`
5	4	`rfib(2)`	`rfib(1) + rfib(0)`
6	5	`rfib(1)`	`return 1`
7	5	`rfib(0)`	`return 0`
8	4	`rfib(1)`	`return 1`
9	3	`rfib(2)`	`rfib(1) + rfib(0)`
10	9	`rfib(1)`	`return 1`
11	9	`rfib(0)`	`return 0`
12	2	`rfib(3)`	`rfib(2) + rfib(1)`
13	12	`rfib(2)`	`rfib(1) + rfib(0)`
14	13	`rfib(1)`	`return 1`
15	13	`rfib(0)`	`return 0`
16	12	`rfib(1)`	`return 1`
17	1	`rfib(4)`	`rfib(3) + rfib(2)`
18	17	`rfib(3)`	`rfib(2) + rfib(1)`
19	18	`rfib(2)`	`rfib(1) + rfib(0)`
20	19	`rfib(1)`	`return 1`
21	19	`rfib(0)`	`return 0`
22	18	`rfib(1)`	`return 1`
23	17	`rfib(2)`	`rfib(1) + rfib(0)`
24	23	`rfib(1)`	`return 1`
25	23	`rfib(0)`	`return 0`

	Wert in `i`	Wert in `a`	Wert in `b`
	-	0	1
	2	1	1
	3	1	2
	4	2	3
	5	3	5
Iterativ:	6	5	8
	7	8	13
	8	13	21
	9	21	34
	10	34	55
	11	55	89
	12	89	144