www.programmieraufgaben.ch

Richtlinien für Autoren

Dieses Dokument beschreibt, wie Aufgaben, Verifikationen und Musterlösungen auf www.programmieraufgaben.ch erfasst werden sollten, damit die Lernaufgaben von Anfängern gut verstanden werden können.
Es handelt sich um grobe Richtlinien, deren Einhaltung nicht zwingend ist; dennoch kann es für Autoren nur von Vorteil sein, die Ziele, Ursprünge und Hintergründe der folgenden Richtwerte zu kennen.

Jede der folgenden Richtlinien erlaubt Ausnahmen, die ganz im Ermessen der Autoren sind.

Es handelt sich also hier nicht um strikte Anforderungen, sondern um eine lose Richtschnur. Wichtig ist mir mehr, dass ein Autor diese Richtlinien kennt. Wenn jemand bessere, striktere oder allgemeingültigere Kriterien konsequent verwendet: umso besser.

Aufgaben: Die Aufgaben müssen erprobt sein. Damit ist gemeint, das mindestens eine Lösung existieren muss. Wenn eine ganze Schulklasse die Aufgabe bereits gelöst hat ist dies sicher von Vorteil. Die Aufgaben müssen in jeder universellen Programmiersprache gelöst werden können. Diese sind so zu formulieren, dass keine Progammiersprachen-abhänige Teile vorkommen.

- Erprobt
- Sprachunabhängig
- In Kapitel eingeordnet (Themenbezug)
- Code als „Pseudocode“

Lösungen: So formulieren, dass keine Progammiersprachen-Tricks verwendet werden. Für den Leser soll sich eine Adaption auf jede andere universelle Sprache möglichst einfach gestalten.

- Funktionsfähigkeit
- Formattierung
- Tricks vermeiden

Verifikationen: Diese sollten für einige typische Eingaben, aber auch für Sonderfälle oder Grenzwerte, die Ausgaben bereitstellen.

Aufgaben

- **Erprobt**: Die Aufgaben sind bereits im Unterricht eingesetzt worden. Mindestens der Referent sollte die Aufgabe bereits gelöst haben, damit die Schwierigkeit abgeschätzt werden kann.
- **Unabhängig** von einer Programmiersprache: Die Aufgaben sollten in verschiedenen Sprachen lösbar sein (C/C++/C#, Perl, Ruby, Java, BASIC, Modula, PL/1, Javascript, BASH, ...).
  Manchmal ist dies nicht möglich, denn ein Teil der Aufgabe wird bereits in einer Sprache vorgegeben. In diesem speziellen Fall ist der Name der Programmiersprache in eckigen Klammern '[ ]' im Aufgabentitel anzugeben. Beispiel: „Türme von Hanoi [Java]“.
  Die Aufgaben sollen jedoch keine Konzepte schulen, die nur in einigen wenigen Sprachen existieren (z. B. Ternärer `:?`-Operator; asserts; lamda-Kalkül…).
- **Themenbezug / Kapitel**: Zu welchen **Kapiteln** gehört eine Aufgabe (Kapitel angeben). Keine vorgreifenden Konzepte verwenden wie. z. B. Arrays im Kapitel über Schleifen; Schleifen im Kapitel über Arrays anzugeben ist natürlich ok. Die Reihenfolge sollte derjenigen im Buch entsprechen. Liegt eine Aufgabe in zwei Kapiteln, so ist das letztere anzugeben: Liegt eine Aufgabe z. B. im Kapitel Schleifen und im Kapitel Felder, so ist ist im Kapitel Felder einzuordnen. Die Reihenfolge ist:
  - Ausdrücke und Datentypen
  - Anweisungen und Abfolgen
  - Selektion (Verzweigung)
  - Schleifen
  - Unterprogramme
  - Felder (Arrays)
  - Zeichenketten (Strings)
  - Datenstrukturen und Sammelobjekte
  - Algorithmen
  - Simulationen
  - Rekursion
- Allfällige Codeteile müssen in **Pseudocode** angegeben werden. Dabei gelten folgende Regeln:
  - Die Zuweisung wird mit dem mathematischen „`:=`“ angegeben.
  - Es existieren nur die folgenden fünf vereinfachten Datentypen `boolean`, `integer`, `real`, `char` und `string`.
  - Datentypen werden nach einem Doppelpunkt hinter die Variable (bzw. den Funktionsprototypen) gestellt (`ok: boolean`; `sin(x: real): real`)
  - Anweisungen einer Sequenz werden jeweils durch eine neue Zeile eingeleitet (Keine Semikolons ';' am Ende der Anweisungen in Pseudocode nötig).
  - Blöcke werden durch geschweifte Klammern '{ }' umgeben. Die Anweisungen in Unterblöcken (Iteration, Subroutine, …) werden eingerückt.
  - Selektionen werden mit `if()`, Iterationen mit `while()` eingeleitet. Andere Kontrollflusssteuerungen sind zu vermeinden, da sie nicht in allen Sprachen gleich vorkommen (insbesondere kein `for()`, `switch()`, …).
    Beispiel umseitig:

```
anzahljahre :=            5
jahr        :=            0
kapital     := startkapital

while(jahr < anzahljahre)
{
  zins      := kapital * zinssatz / 100
  kapital   := kapital + zins
  kapital   := startkapital
  jahr      := jahr + 1
}
endkapital  := kapital
```

Lösungen

Funktionsfähigkeit: Lösungen sollten nach dem Herunterladen ohne Anpassungen funktionieren.

Die **Formattierung** von Code sollte so gestaltet sein, wie es in der aktuellen Sprache üblich ist: Einrücken, Groß-/Kleinschreibung, Namenswahl, Variablenbezeichnungen etc.; am besten nach Vorgabe der Sprach-Entwickler (java: sun/oracle; c: K&R; VB: Microsoft; …)

Programmiersprachen-Tricks: Generell gilt, dass Programmcode von Lösungen so entwickelt werden sollte, dass dieser Code auch **verstanden** wird, wenn der Leser die aktuelle Sprache **nicht** vollständig beherrscht. So ist es möglich, Musterlösungen einfach in andere Sprachen zu übersetzen.

Besser so (Sprachunabhängig):	*So nur bedingt (wenn möglich vermeiden):*
Um das Konzept der Variable besser einzugehen soll erklärt werden, was $x_{i+1} := x_i + 1$ bedeutet und danach folgende (linke) Schreibweise verwendet werden. Auf ein „`++`“-Operator und dergleichen (`inc()`) ist zu verzichten, denn er ist nicht unabhängig von einer Sprache:	
`x := x + 1`	`x++;`
Gängige Konzepte (wie z. B. die for-Schleife) sind legitim.	
`for(i := 1; i <= 10; i := i + 1)` `FOR i := 1 TO 10 DO`	
Auf break und continue wenn möglich zugunsten einer Selektion verzichten:	
`if(…) {…} else {…}` `if(…) {…} else {…}`	`break;` `continue;`
Sicheren Code verwenden, der in anderen Sprachen nicht zu Fehlverhalten (z. B. hier einer Zuweisung) führen kann:	
`if(a) {…} // a: boolean`	`if(a = true) {…}`
`if(0 = i) {…}`	`if(i = 0) {…}`
Sprachunabhängige Syntax ist vorzuziehen. Natürlich darf aus Überlegung der Einfachheit (Simplicity) die Lösung rechts auch gezeigt werden.	

`if(a > b) {`
`  x := a`
`} else { // a <= b:`
`  x := b`
`}`
	`x = a > b ? a : b;`

`x := x * 256`	`X <<= 8;`

`bt := read()`
`while(-1 <> bt) {`
`  …`
`  bt := read()`
`}`
	`while(-1 != (bt = read()))`
	`{`
	`  …`
	`}`

Besser so (Sprachunabhängig):	*So nur bedingt:*
Wenn eine Programmiersprache eine Aufgabe mit einer Zeile lösen kann (API-Aufruf), so ist dies sicher ein legitimer – und auch als Musterlösung wünschenswerter – Weg. Für eine allgemeingültigkeit der Lösung (wenn auch nicht zwingend) ist es ebenso wünschenswert zu zeigen, wie die aufgerufene API-Prozedur funktioniert. Solches ist dann auch einfach in eine andere Sprache zu übersetzen.	

`// core Java:`
`wort = wort.reverse();`
allgemeine Lösung:
`// pr  = Position ab rechts`
`// pos  = Pos. von links.`
`// wort = Original`
`// rev  = Umkehrung`
`int pos, pr;`
`pos = 0;`
`while(pos < wort.length())`
`{`
`  pr  = wort.length() - pos - 1;`
`  rev = rev + wort.charAt(pr);`
`  pos = pos + 1;`
`}`
	`wort = wort.reverse();`

So selten wie möglich statischen Code und Programmiersprachen abhängige Schlüsselwörter (wie hier `public`) verwenden. Natürlich ist das in einigen Sprachen (wie z. B. Java) nicht vollständig zu vermeiden. Es lässt sich jedoch für sprachunabhängige Aufgaben oft auf ein einziges Vorkommen (`main()`) reduzieren.	
`void a()`	`public static void a()`

Verifikationen

Der Verifikationsteil ist dann nötig, wenn die Ausgabe eines Programms nicht offensichtlich ist. Die Verifikation soll für einige wenige Eingaben die korrekte Ausgabe angeben. Dies ist nicht nötig, wenn die Ausgabe trivial ist. Bei einigen Aufgaben reicht als Verifikation auch die Anzahl der Lösungen, oder ein Teil der Ausgabe völlig aus.
Spezialfälle (Division durch Null, gerade Primzahlen, Wurzeln aus negativen Zahlen, unzulässiger Input, …) sollten hier in den Verifikationen berücksichtigt werden. Die Lernenden sind oft froh, wenn das Programm für einfache Testdaten läuft und vergessen die üblichen Sonderfälle.

Dokumentversion: 6. Nov. 2012
Autor: philipp gressly freimann