TEKO
SCHWEIZERISCHE FACHSCHULE

Programmstrukturen

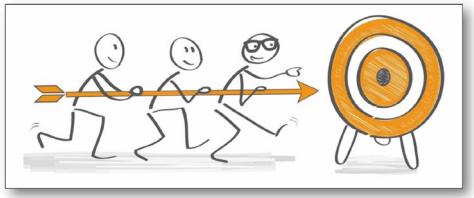
Björn Näf

Dozent

bjoern.naef@edu.teko.ch

Ziele: Programmstrukturen

6 Lektionen



◢ Lernfelder / Lerninhalte

- Grundlegende Strukturen (Sequenz, Selektion und Iteration)

◢ Lernziele

- Sie kennen den Programmaufbau, sie können einfache lineare Programme selbstständig analysieren und korrigieren.
- Sie kennen die Selektion und die dazugehörigen Komponenten. Sie können Programme entwerfen, in welchen mehrfache sowie verschachtelte Verzweigungen erforderlich sind.
- Sie kennen die Iteration und können kopf- sowie fussgesteuerte Iterationen programmieren.

Agenda

TEKO
SCHWEIZERISCHE FACHSCHULE

- **Warm Up: „Programmieren“ mit Diagrammen**
- **Programmstrukturen**
  - **Sequenz**
  - **Selektion**
  - **Iteration**

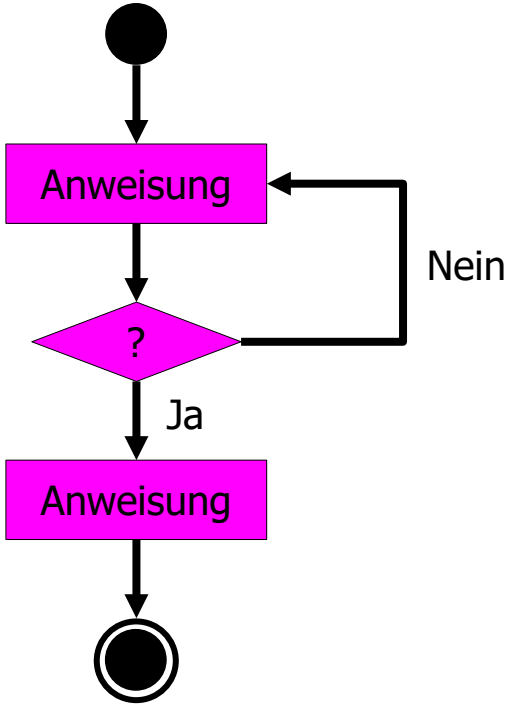
Fluss- / Aktivitätsdiagramm als Modell der Programmstruktur

«Programmieren mit Ablaufdiagrammen»

Diagramm, welches Ablauf oder Prozess darstellt

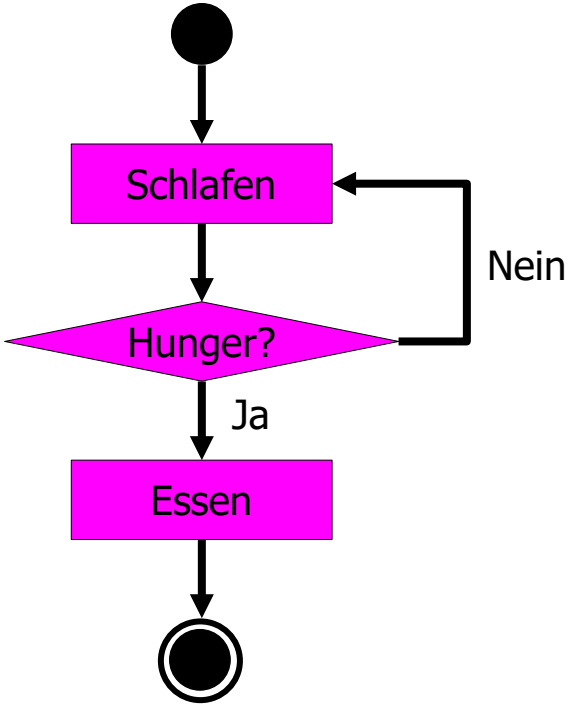
- zeigt Anweisungssequenzen
- kann Verzweigungen enthalten
- kann damit auch Wiederholungen darstellen

Bemerkung: Das Flussdiagramm ist eine leichtgewichtigere (weniger «strenge») Variante des Aktivitätsdiagramms



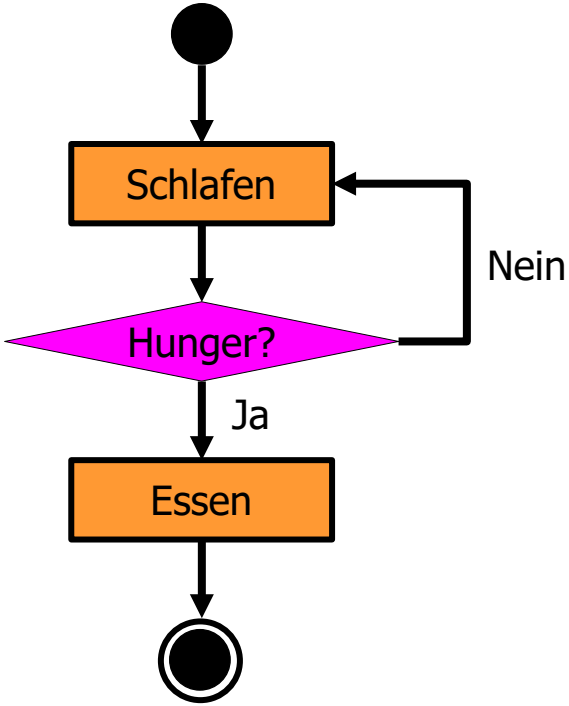
Programmstruktur: Beispiel

- Beispiel aus dem Alltag



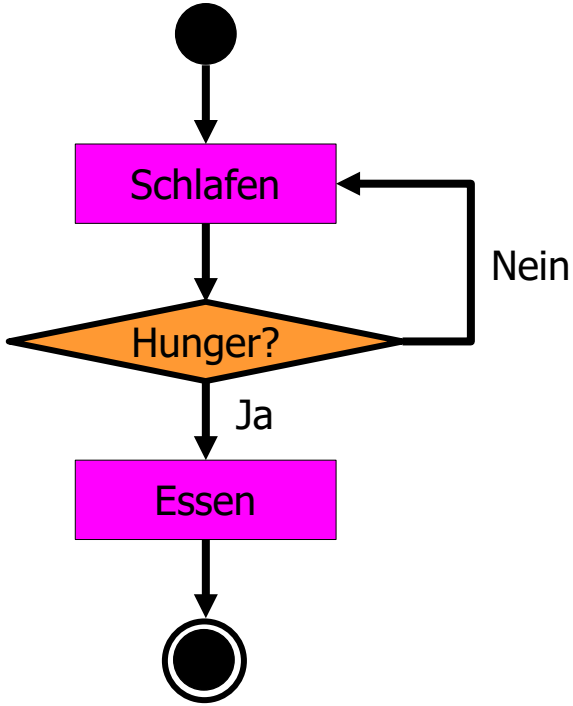
Programmstruktur: Anweisungen

- Sequenz von Operationen und Tätigkeiten



Programmstruktur: Entscheidungen

- Verzweigung in Abhängigkeit einer Bedingung

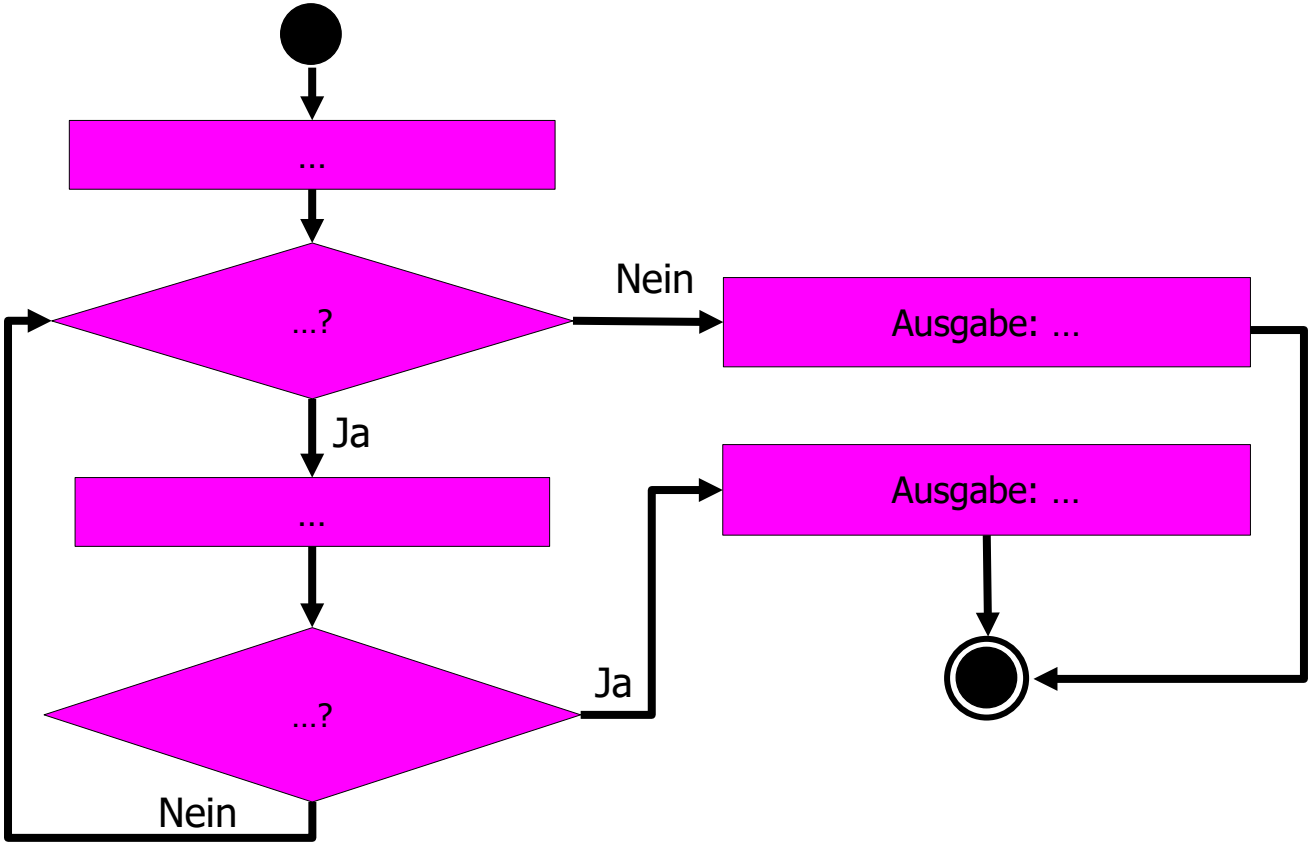


Programmstruktur: Beispiel

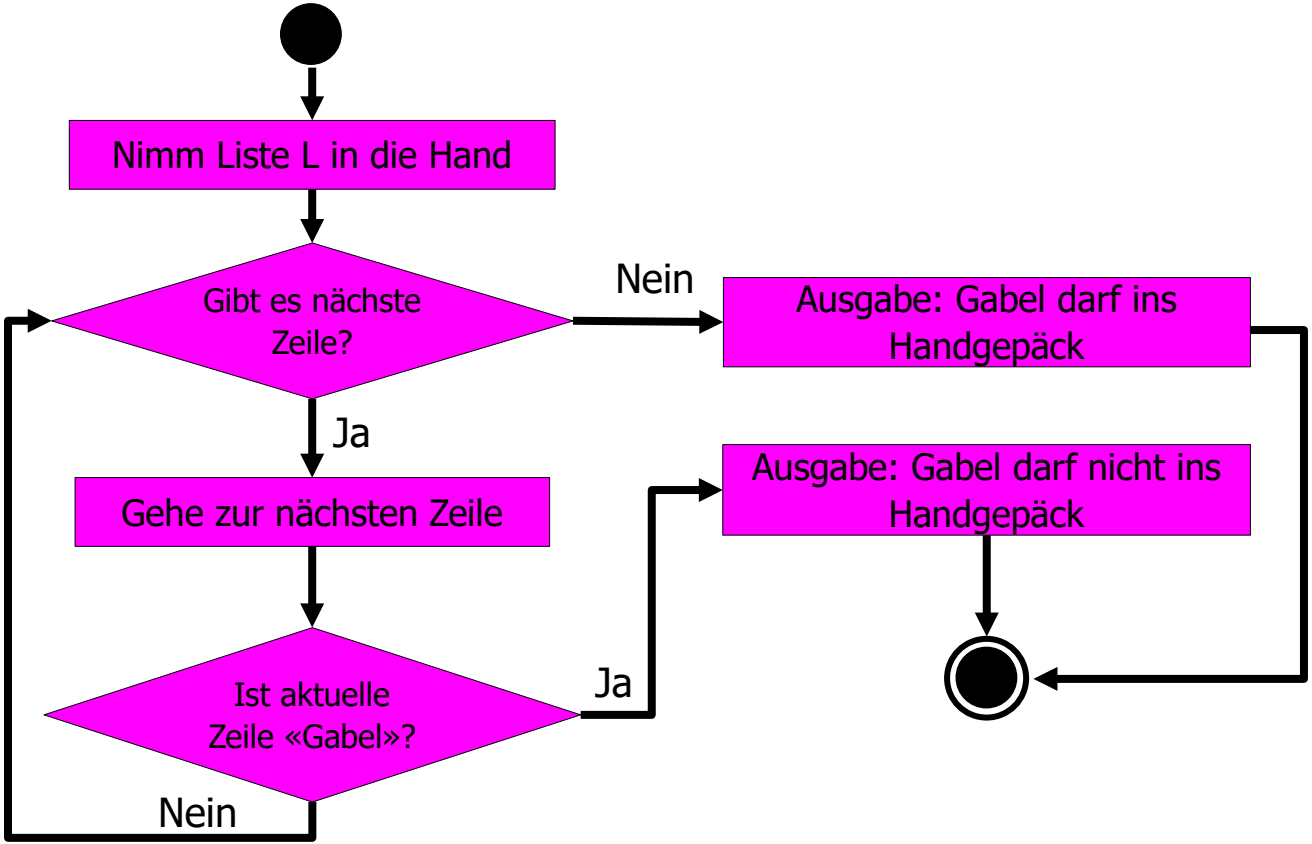
«Trockenübung»

- Gegeben sei eine Liste verbotener Gegenstände.
  Darf eine Gabel im Handgepäck auf einen Flug mitgebracht werden?

Programmstruktur: Beispiel Vorlage



Programmstruktur: Beispiel Lösung

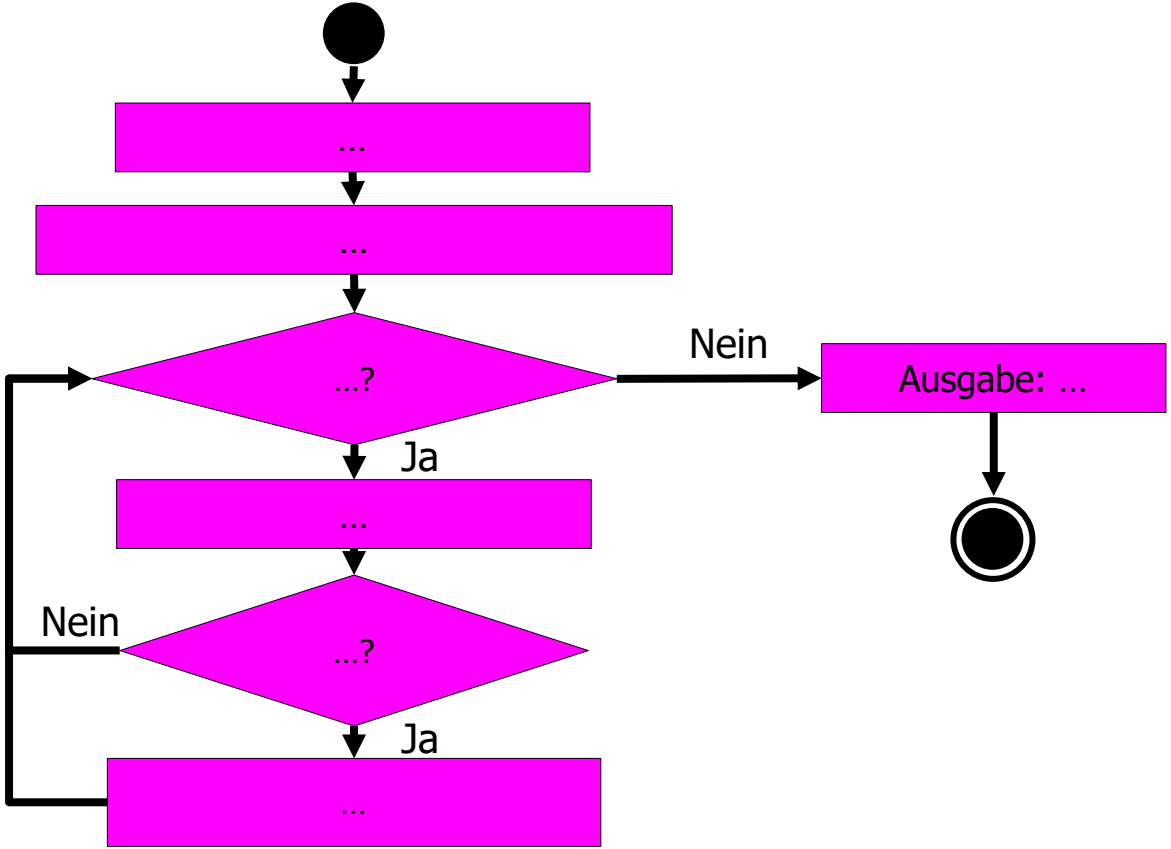


Aufgabe 2.1

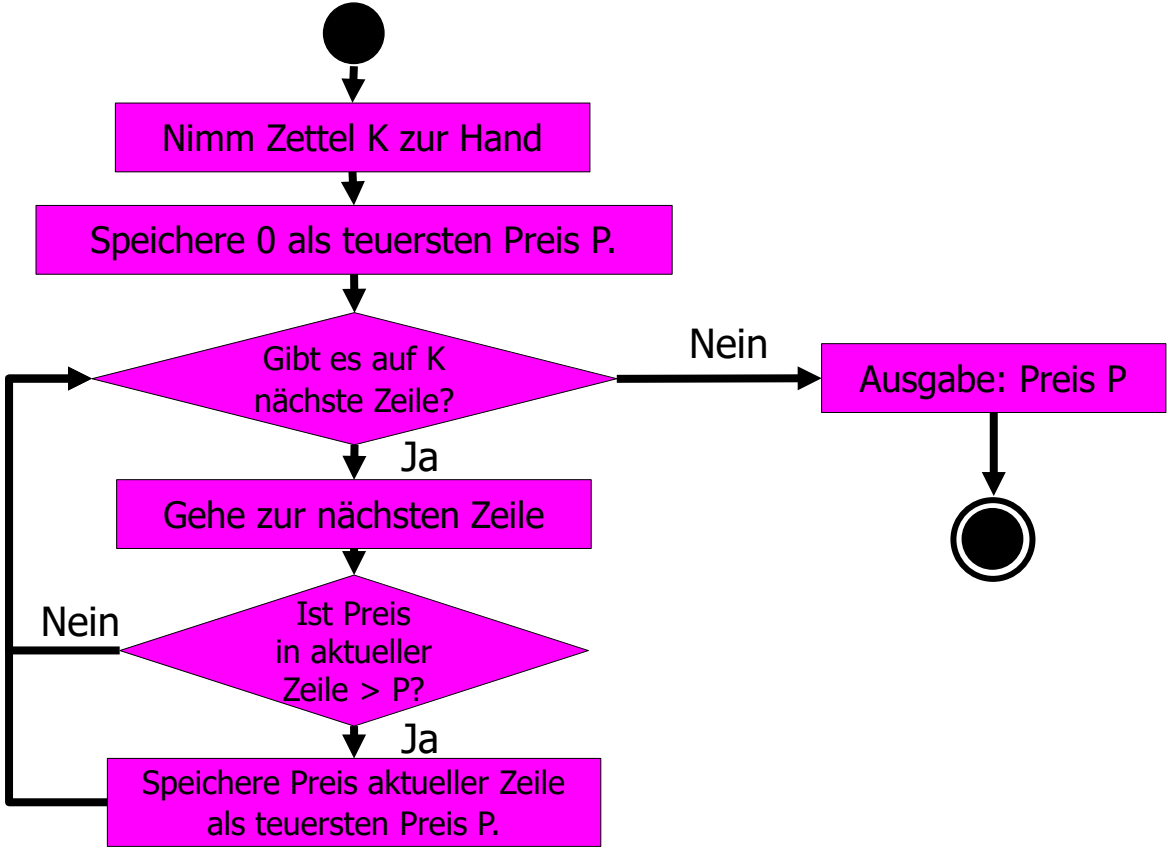
Programmstruktur mittels Flussdiagramm modellieren

- Gegeben sei ein Kassenzettel K aus dem Supermarkt.
  Wieviel hat das teuerste Lebensmittel gekostet?

Aufgabe 2.1: Vorlage



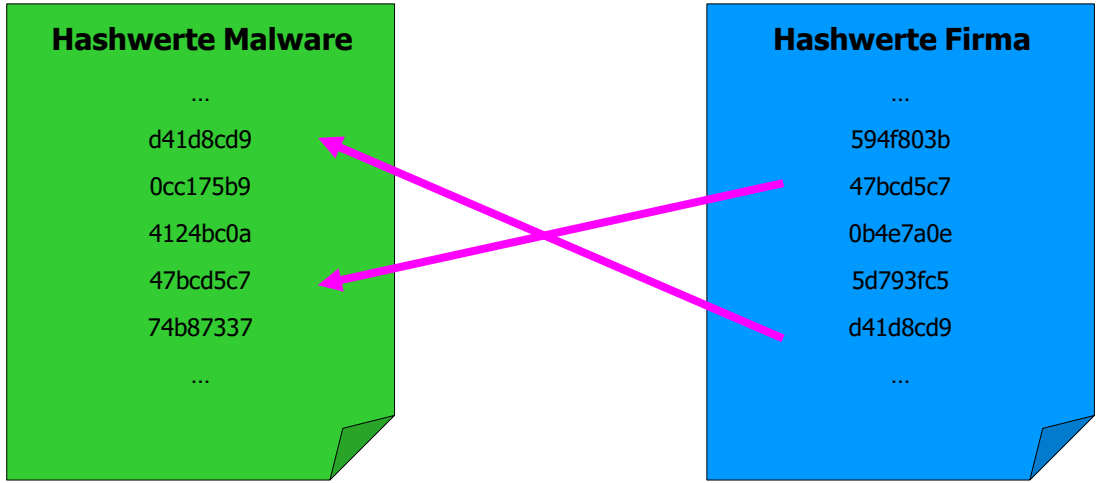
Aufgabe 2.1: Lösung



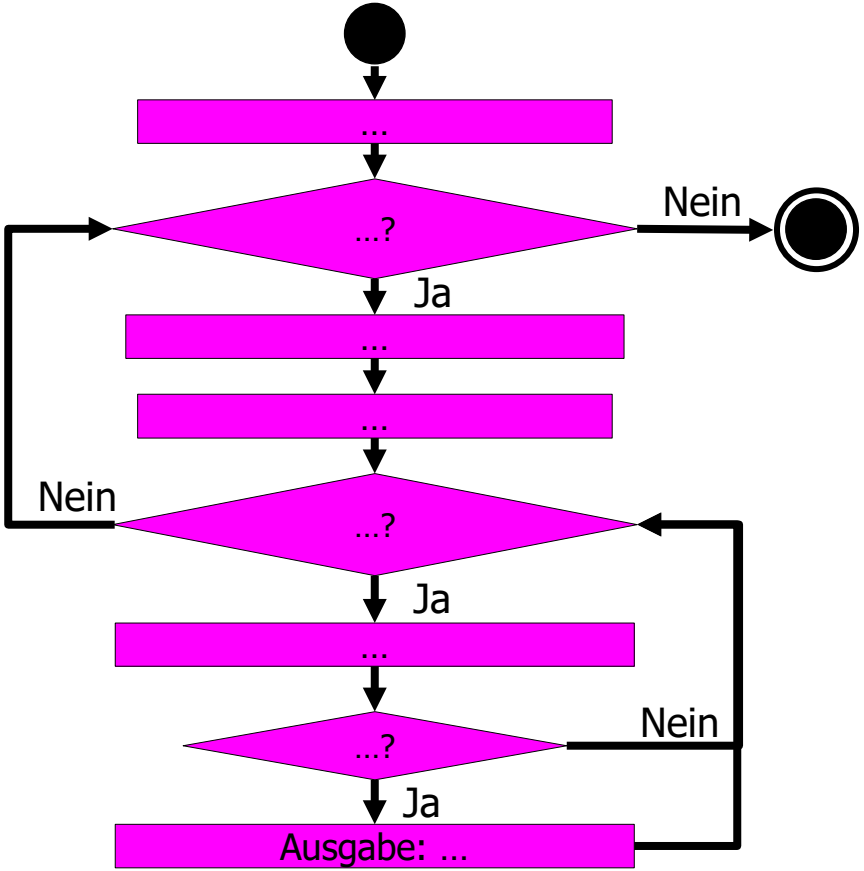
Aufgabe 2.2

Programmstruktur mittels Flussdiagramm modellieren

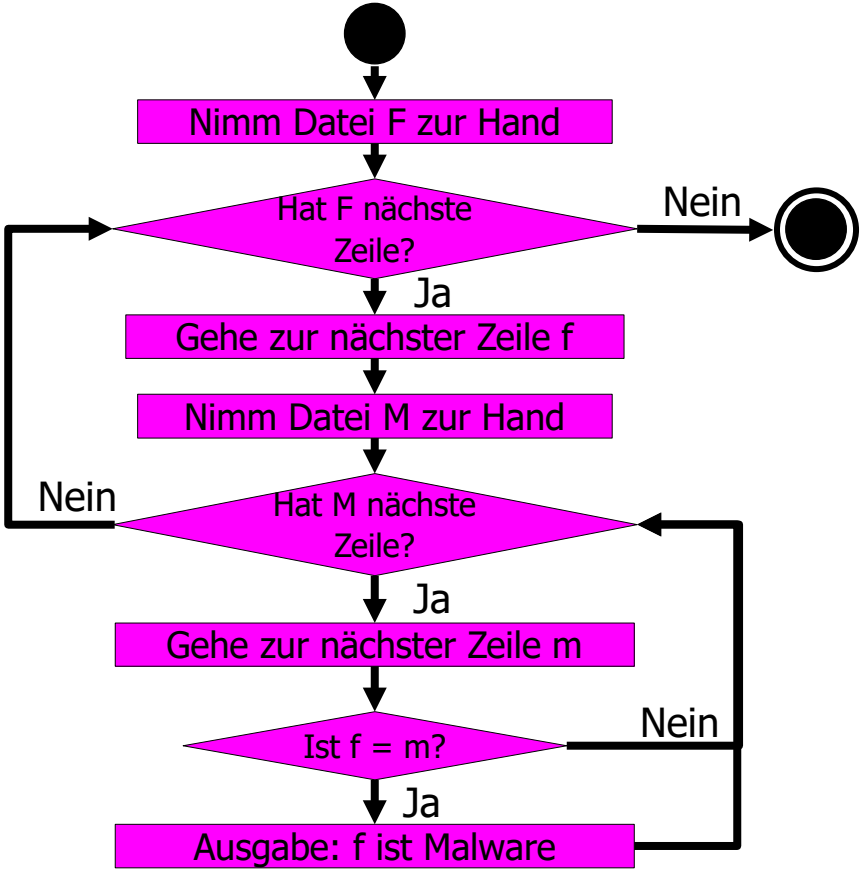
- Nach Hackerangriff: Gegeben eine Datei M mit Hashwerten von Malware und eine Datei F mit Hashwerten von Dateien einer Firma. Welche Dateien der Firma sind Malware?



Aufgabe 2.2: Vorlage



Aufgabe 2.2: Lösung



Was wir bisher gelernt haben

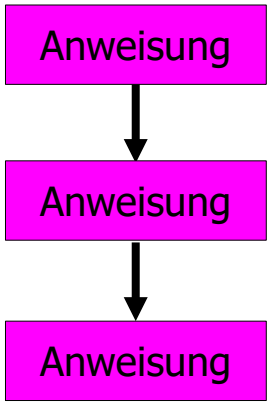
TEKO
SCHWEIZERISCHE FACHSCHULE

- Fluss- oder Aktivitätsdiagramme stellen Prozesse bzw. Abläufe dar
- Elemente eines Fluss- oder Aktivitätsdiagrammes:
  - Start und Ende eines Prozesses
  - Anweisungen (u.a. Ausgabeanweisungen)
  - Verzweigungen (auch: Bedingungsprüfungen, Selektionen)
  - Schleifen (auch: Wiederholungen, Schlaufen, Iterationen)
  - Anweisungen, Verzweigungen und Schleifen werden als Sequenz verknüpft (=chronologischer, linearer Ablauf als Prozess)

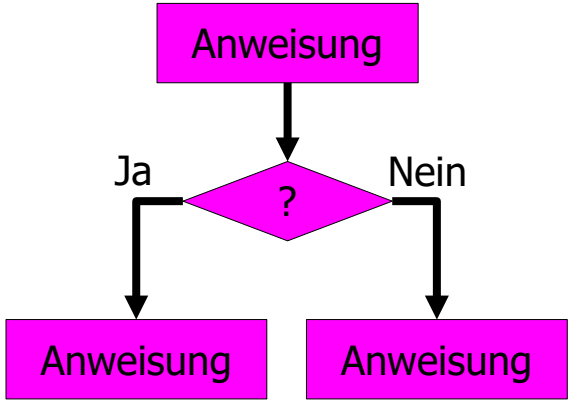
Programmstruktur: Kontrollstrukturen

Umsetzung eines Ablaufs in einer Programmiersprache

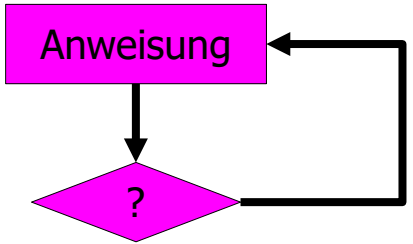
In jeder Programmiersprache gibt es drei grundlegende Kontrollstrukturen:



Sequenz
(Aneinanderreihung)



Selektion
(Entscheidung/Verzweigung)



Iteration
(Wiederholung/Schleife)

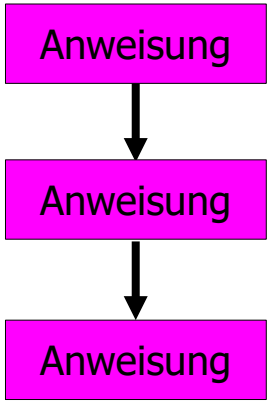
Anweisungen (Statements)

TEKO
SCHWEIZERISCHE FACHSCHULE

- Programm-“Befehle“
- Einzel-Anweisungen werden mittels Zeilenschaltung oder Semikolon ; abgeschlossen / getrennt
- Code-Blöcke { } kapseln zusammengehörige Anweisungen der gleichen Programmebene
- Spezielle Anweisungen steuern den Kontrollfluss:
  - Selektion
  - Iteration
  - etc.

Sequenz

Lineare, bedingungslose Aneinanderreihung von Anweisungen




```
val a = 5
val b = 8
val c = 3

val s = a + b + c

println(“Summe: “ + s)
```

Aufgabe 2.3

Sequenz

1. Lesen Sie von der Tastatur nacheinander 3 Zahlen ein. Verwendung Sie

```
val zahlAlsString = readln()       // zahlAlsString durch Variablenname ersetzen
```

2. Wandeln Sie die eingegebenen Werte in Ganzzahlen um:

```
val zahl = zahlAlsString?.toIntOrNull()   // zahl / zahlAlsString ggf. ersetzen
```

3. Berechnen Sie die Summe der drei Zahlen und geben Sie das Ergebnis auf der Konsole aus

```
// Das haben Sie bereits gelernt ;)
```

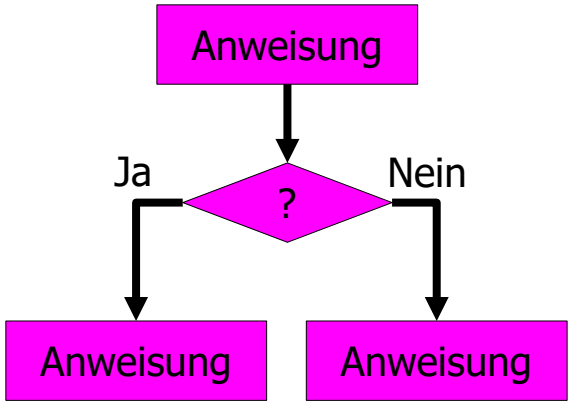
4. Was passiert, wenn anstatt einer Zahl ein Buchstabe, ein Text oder gar nichts eingegeben wird?

```
// Verwenden Sie allenfalls das Debugging, um Details herauszufinden
```

Selektion

An eine Bedingung gebundene Entscheidung / Verzweigung

- Verzweigungen im Code erfolgen auf Basis eines bool'schen Ausdrucks (true / false)
- Nur ein Ast (Code-Block) wird beim Ansprechen der Selektion ausgelöst



Umsetzungsvarianten in Kotlin:

- **if / else if / else**

  → ideal für 1-3 Fälle oder logische Entscheidungen

- **when** (heisst in anderen Sprachen „switch / case")

  → ideal für 3+ Fälle mit festen Werten

Selektion einfach

Bedingte Ausführung eines Code-Bocks mit « if »

if ([Bedingung])
{
[Anweisungen]
}

oder (nicht empfohlen):

if ([Bedingung])
[Einzelanweisung]

{ } können weggelassen werden, wenn der Code-Block nur eine einzige Anweisung umfasst

```
val my_zahl = 5

if (my_zahl > 4)
{
    println("Zahl ist grösser 4")
}
```

```
val my_zahl1 = 5; val my_zahl2 = -3;

if (my_zahl1 > 4)
{
    if (my_zahl1 < 0)
    {
        println("Bedingungen erfüllt")
    }
}
```

Selektion «erweitert»

Alternative Ausführung zweier Code-Blöcke anhand Bedingung mit « if-else »

```
if ( [Bedingung] )
{
   [Anweisungen]
}
else
{
   [Anweisungen]
}
```

```
val my_zahl = 3

if (my_zahl > 0)
{
   //Block 1
   println("Zahl ist grösser 0")
}
else
{
   //Block 2
   println("Zahl ist kleiner als 0")
}
```

Selektion «vollständig»

« if - else if - else»

```
if ( [Bedingung1] )
{
   [Anweisungen]
}
else if ( [Bedingung2] )
{
   [Anweisungen]
}
else
{
   [Anweisungen]
}
```

```
val my_zahl = 1

if (my_zahl > 1)
{
   //Block 1
   println("Zahl ist grösser 4")
}
else if (my_zahl < 1)
{
   //Block 2
   println("Zahl ist kleiner als 4")
}
else
{
   //Block 3
   println("Zahl ist gleich 4")
}
```

Geschachtelte Selektion

Häufige Ursache für spät erkannte Bugs!

Code-Blöcke können geschachtelt werden (in Selektionen, Iterationen etc.). Beispiel:

```
if ( x % 2  == 0 )
   if ( x < 0 )
       println ( "${x} ist eine gerade, negative Zahl" )
else
   println ( "${x} ist eine ungerade Zahl" )
```

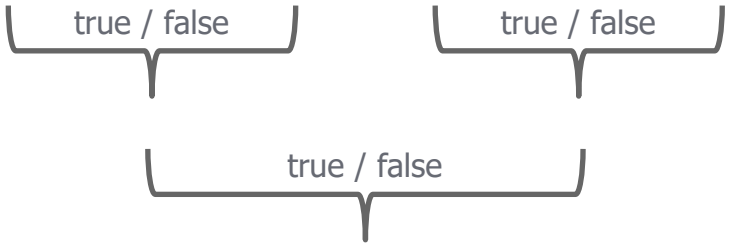
- Was wird für x=4 angezeigt?
- **Wie würden Sie das Problem lösen?**

Selektion mit logischen Verknüpfungen

Die Bedingung kann ein zusammengesetzter bool'scher Ausdruck sein

```
if ( [Bedingung1] [Operator] [Bedingung2] )
```

```
if ( my_zahl > 4  &&  my_zahl < 10 )
```



Die wichtigsten Operatoren

Repetition

Operator	Bedeutung	Beispiel
`<`	Kleiner als	`zahl < 5`
`<=`	Kleiner als oder gleich	`zahl <= 0`
`>`	Grösser als	`zahl1 > zahl2`
`>=`	Grösser als oder gleich	`zahl1 >= zahl2`
`==`	Gleichheit (arithmetisch, logisch, referenziell)	`zahl == 0`
`!=`	Ungleichheit	`zahl1 != zahl2`
`&&`	Logisches UND	`zahl > 0 && zahl <= 10`
`||`	Logisches ODER	`zahl1 == 0 || zahl2 == 0`
`!`	Logisches NICHT	`!(zahl == 0)  // => zahl != 0`

Aufgabe 2.4

Selektion mit « if »

Erweitern Sie das Programm aus Aufgabe 2.3 um folgende Selektionen mit «if»:

1. Der erste eingegebene Wert muss grösser als oder gleich 5 sein.
2. Der zweite eingegebene Wert darf nicht gleich 10 sein.
3. Der dritte eingegebene Wert muss grösser als 5 und darf nicht grösser als 10 sein.

Geben Sie mit den Ihnen bereits bekannten Anweisungen eine Fehlermeldung aus, falls (mindestens) eine der drei Bedingungen nicht erfüllt ist. In der Fehlermeldung muss ersichtlich sein, warum die Eingabe die jeweilige Bedingung nicht erfüllt.

Testen Sie Ihr Programm mit verschiedenen Eingabekombinationen.
Verwendung Sie den Debugger, um allfällige Programmfehler zu finden.

Selektion mit «when» - Prüfung auf Gleichheit

Bedingte Ausführung eines Code-Bocks anhand bestimmter, fester Werte

```
when ( [Variable] )
{
    [Wert1] ->
      [Anweisungen]

    [Wert2] ->
      [Anweisungen]

    [...]

    else ->
      [Anweisungen]
}
```

```
val zahl = 4

when (zahl)
{
        1 -> println("Sie haben Eins eingegeben.")
        2 -> println("Sie haben Zwei eingegeben.")
        3 -> println("Sie haben Drei eingegeben.")
        else -> println("Ungültige Eingabe.")
}
```

Funktioniert nur mit einfachen Datentypen:
Int, Long, Short, Byte, Char, String, Boolean
sowie Enum-Klassen

Selektion mit «when» - Prüfung auf Wertebereich

Bedingte Ausführung eines Code-Bocks anhand eines vorgegebenen Bereichs

```
when ( [Variable] )
{
    in [Wert11]...[Wert12] ->
      [Anweisungen]

    in [Wert21]...[Wert22] ->
      [Anweisungen]

    [...]

    else ->
      [Anweisungen]
}
```

```
val zahl = 42

when (zahl) {
    in 1..10 -> println("Die Zahl liegt zwischen 1 und 10.")
    in 11..20 -> println("Die Zahl liegt zwischen 11 und 20.")
    else -> println("Die Zahl liegt ausserhalb des Bereichs.")
}
```

Selektion mit «when» - Prüfung auf Objekttyp

Bedingte Ausführung eines Code-Bocks anhand eines Datentyps

```
when ( [Variable] )
{
    is [Datentyp] ->
      [Anweisungen]

    is [Datentyp] ->
      [Anweisungen]

    [...]

    else ->
      [Anweisungen]
}
```

```
val irgendwas: Any = "Hallo, Welt!"

when (irgendwas) {
    is String -> println("Es ist ein String.")
    is Int -> println("Es ist eine ganze Zahl.")
    else -> println("Es ist etwas anderes.")
}
```

Aufgabe 2.5

Selektion mit « when »

Erweitern Sie das Programm aus Aufgabe 2.4, indem Sie zur Überprüfung des **dritten** eingegebenen Wertes eine Selektion mit «when» einbauen:

1. Der dritte eingegebene Wert darf nicht gleich 7 sein.
2. Der dritte eingegebene Wert darf nicht zwischen 10 und 20 liegen.

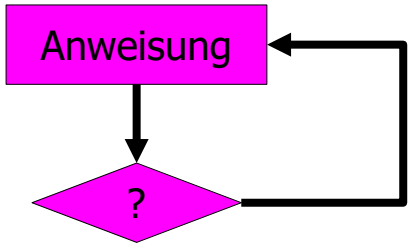
Geben Sie eine Fehlermeldung aus, falls eine der Bedingungen nicht erfüllt ist.

Testen Sie Ihr Programm mit verschiedenen Eingabekombinationen.
Verwendung Sie den Debugger, um allfällige Programmfehler zu finden.

Iteration

An eine Bedingung gebundene Wiederholung (Schleife / Schlaufe)

- Code-Block wird wiederholt ausgeführt, solange oder bis eine Bedingung erfüllt ist.
- Bedingung kann explizit («Mache solange bis») oder implizit («Mache für alle Elemente») sein
  - Je nach Iterationstyp wird ein logischer Ausdruck oder ein Zähler verwendet
- Bedingung kann (teilweise) vor oder nach dem Schleifendurchlauf stehen (Kopf-/Fuss-Steuerung)



- **Zählschleife** „for“

  → Anzahl Durchläufe vorab bestimmbar

- **Mengenschleife** „for “

  → Durch eine ganze Liste / Kollektion / Map iterieren

- **Kopf-/fussgesteuerte Bedingungsschleife** „while“ / „do-while“

  → Logische Abbruchbedingung

Iteration: Zählschleife « for »

verwendet, wenn klar ist, wie oft die Schleife durchlaufen werden soll

for ([Variable] in [Startwert]..[Endwert] step [Schrittweite]) // Schrittweite ist optional
{
[Anweisungen] //Schleifenkörper
}

```
for (i in 1..5)
{
    println(i)
}
```

oder in absteigender Reihenfolge:

for ([Variable] in [Startwert] downTo [Endwert])
{
[Anweisungen] //Schleifenkörper
}

```
for (i in 10 downTo 1 step 2) {
    println(i)
}
```

Achtung: Die (Zähl-) Variable ist nur innerhalb des Schleifenkörpers gültig (“Scope”)

Iteration: Mengenschleife « for »

Verwendet, um alle Elemente einer Liste oder Map nacheinander zu durchlaufen

```
for ( [Listenelement] in [Liste] )
{
   [Anweisungen]
}
```

```
val namen = listOf("Alice", "Bob", "Charlie")

for (name in namen)
{
   println(name)
}
```

oder für Map:

```
for ( [Schlüsselvariable, Wertvariable] in [Map] )
{
   [Anweisungen]
}
```

```
val map = mapOf("Alice" to 25, "Bob" to 30, "Charlie" to 35)

for ((name, alter) in map)
{
   println("$name ist $alter Jahre alt.")
}
```

Aufgabe 2.6

Iteration mit « for »

Ändern Sie das Programm aus Aufgabe 2.5 so, dass der Benutzer selber wählen kann, wie viele Zahlen eingegeben und anschliessend summiert werden sollen.

1. Verwenden Sie für die Eingabewerte eine Liste und eine Zählschleife als Iteration.
2. Ersetzen Sie anschliessend die Zählschleife durch eine Mengenschleife.

Wo liegen die Hauptunterschiede bei der Verwendung dieser beiden Iterationstypen?

Iteration: Bedingungssschleife kopf- und fussgesteuert « while »

Verwendet, um Anzahl Wiederholungen an logische Bedingung zu binden

while ([Bedingung])
{
[Anweisungen]
}

wird im Grenzfall nie durchlaufen
(Zuerst wird geprüft und dann ausgeführt)

```
var zahl = 0

while (zahl < 5)
{
   println(zahl)
   zahl++
}
```

do
{
[Anweisungen]
} while ([Bedingung])

wird im Grenzfall 1x durchlaufen
(Zuerst wird ausgeführt und dann geprüft)

```
var zahl = 0

do
{
   println(zahl)
   zahl++
} while (zahl < 5)
```

Aufgabe 2.7

Iteration mit « while » und « do-while »

Ersetzen Sie die Iteration im Programm aus Aufgabe 2.6:

1. Verwenden Sie eine kopfgesteuerte Schleife («while»).
2. Verwenden Sie eine fussgesteuerte Schleife («do-while»).

Wo liegen die Hauptunterschiede bei der Verwendung dieser beiden Iterationstypen?

Wo liegen die Hauptunterschiede zur Verwendung der Zähl- und der Mengenschleife?

Häufige Fehler bei Selektionen und Iterationen

Kennen heisst vermeiden können

- **Bedingung** falsch formuliert, z.B.
  - Falsche Operatoren (z.B. > statt >=)
  - Operatorpräzedenz nicht beachtet → Klammern () setzen
  - Präzisionsverlust nicht beachtet (z.B. Test auf Gleichheit bei Gleitkommazahlen)

- **Verschachtelung** falsch → Klammern { } setzen

- **Variablen-Gültigkeit** nicht beachtet → falsches Ergebnis → Debuggen

Häufige Fehler bei Selektionen und Iterationen (2)

Kennen heisst vermeiden können

Spezifisch bei Selektionen:

- **if-elseif-else-Logik** falsch kombiniert → Debuggen, Klammern setzen

Spezifisch bei Iterationen:

- **Initialisierung** falsch → Anzahl Durchläufe und/oder Ergebnis inkorrekt
- **Schleifentyp** falsch gewählt (z.B. kopf- vs. fussgesteuert) → Anzahl Durchläufe
- Endlosschleife → Abbruchbedingung prüfen / korrigieren, Debugging

Fehler in Selektionen und Iterationen finden

Grenzfälle identifizieren und als Testfälle definieren

Selektionen und Iterationen testen mit Eingabewerten, die

- in jedem Fall wahr oder falsch sind / sein sollen
- null Iterationen auslösen sollen
- genau eine Iteration auslösen sollen
- die maximal mögliche Anzahl Iterationen auslösen sollen
- die maximal mögliche Anzahl Iterationen übertreffen sollen
- Etc.

Debugger verwenden und Variablen-Werte beobachten!

Last but not least

Übung macht den Meister!

TEKO
SCHWEIZERISCHE FACHSCHULE

“ Good programming comes from experience, experience comes from bad programming ”

Zusatz-Material

Unäre Inkrement- und Dekrement-Operatoren

TEKO
SCHWEIZERISCHE FACHSCHULE

Operatoren zum Addieren oder Subtrahieren von 1 von/zu einer numerischen Variablen

Operator	Bezeichnung	Semantik
++a	Präfix-Inkrement	Erhöht um 1 und verwendet danach den neuen Wert
a++	Postfix-Inkrement	Verwendet den aktuellen Wert und erhöht danach um 1
--b	Präfix-Dekrement	Reduziert um 1 und verwendet danach den neuen Wert
b++	Postfix-Dekrement	Verwendet den aktuellen Wert und reduziert danach um 1

Wahrheitstabelle

A	B	A || B	A && B
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

A	! A
true	false
false	true

De Morgan Regeln

Umformung logischer Ausdrücke (z.B. in Selektionen oder Iterationen)

Die folgenden Bedingungen sind logisch äquivalent:

```
( ! (num >= 0 && num <= 100) )

( num < 0 || num > 100 )
```

Verallgemeinerung: De Morgan Regeln:

```
! (a   &&   b)  =   !a   ||   !b
! (a   ||   b)  =   !a   &&   !b
```