

## 3

## Logische Abfragen, bedingte Anweisungen und logische Operatoren

Logische Abfragen sind im Grunde das einzige Mittel, das uns für die Programmierung zur Verfügung steht, da wir nur die Möglichkeit haben, Speicherzellen miteinander zu vergleichen. Alle Befehle oder Programmiermethoden wie Schleifen und Funktionen sind im Wesentlichen nur geschickte Kombinationen logischer Abfragen. Wir können dabei sogar nur überprüfen, ob eine Speicherzelle, die wir im Kapitel Variablen mit Miniakkus verglichen haben, »gefüllt« oder »leer« ist. Wir haben bei allen modernen Programmiersprachen aber den Vorteil, dass eine ganze Reihe nötiger Systemüberprüfungen oder Abfragen automatisch »im Hintergrund« vorgenommen werden, wenn wir logische Abfragen programmieren. Es werden automatisch Speicherbereiche für Variablen festgelegt oder Variablen bitweise miteinander verglichen. Bei den Ursprüngen der Programmierung, der Maschinensprache bzw. dem Assembler muss man sozusagen noch alles selbst programmieren.

### Wie könnten wir uns eine logische Abfrage technisch vorstellen?

Wir wollen zum Beispiel überprüfen, ob der (Programm)Benutzer die linke Maustaste gedrückt hat. Dazu benötigen wir eine Speicherzelle, die gefüllt (mit 1 beschrieben) wird, wenn der Benutzer die linke Maustaste drückt, und geleert oder gelöscht (mit 0 beschrieben) wird, wenn die Maustaste nicht gedrückt ist oder losgelassen wurde. Das kann man sehr leicht mit elektronischen Schaltungen lösen. Diese Speicherzelle können wir mit einem Messgerät überprüfen. *Wenn* am Messgerät die Kontrollleuchte angeht, ist die Speicherzelle gefüllt, *dann* ist also die Maustaste *gedrückt*, *und* wenn die Kontrollleuchte nicht angeht, ist die Maustaste nicht gedrückt. In der Praxis sieht das natürlich etwas anders (komplizierter) aus, aber für das Verständnis darf oder kann uns diese Anschauung genügen.

Wir erkennen aber hier schon deutlich die logische Abfragestruktur *wenn/dann*, und da es bei der Programmierung international zugeht, werden logische Abfragen als *if/then*-Abfragen bezeichnet.

*Wenn (if)* die Maustaste gedrückt wird, *dann (then)* leuchtet die Kontrollleuchte an unserem Messgerät. Ist die Maustaste aber nicht gedrückt, bleibt auch die Kontrollleuchte dunkel.

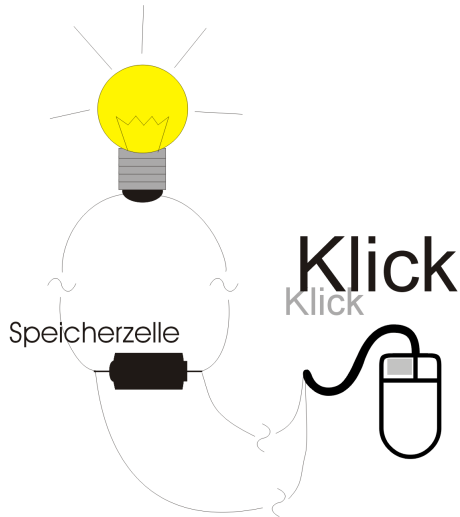


Abbildung 3.1: Bildhafte if/then-Überprüfung

### 3.1 Die if/then-Abfrage

Eine der häufigsten verwendeten logischen Abfrage beim Programmieren ist die **if/then**-Abfrage. Mit dieser Abfrage werden Werte, Variablen(inhalte) oder Zustände miteinander verglichen. Bei diesem Vergleich erhalten wir ein Ergebnis. Entweder stimmt das Ergebnis des Vergleichs (der Abfrage), oder es stimmt nicht. Wenn der Vergleich stimmt, bekommen wir als Ergebnis die Bestätigung, dass der Vergleich stimmt oder **wahr** (*true*) ist. Stimmt der Vergleich nicht, dann ist das Ergebnis **falsch** (*false*).

Wir können es uns etwa so vorstellen, dass das Ergebnis des Vergleichs der if/then-Abfrage vom System zur Kontrolle in einer Speicherzelle als eine Art lokale Variable gespeichert (gesichert) wird. Beim Ergebnis **wahr** ist diese Speicherzelle mit dem Wert **1** (*true*) belegt und wenn das Ergebnis **falsch** ist, hat die Speicherzelle den Wert **0** (*false*). Diese lokale Variable wird nur für diese logische Abfrage verwendet und anschließend wieder gelöscht.

Wir brauchen aber nicht selbst die Speicherzelle für das Ergebnis des Vergleichs zu kontrollieren (auszulesen), sondern müssen lediglich bestimmte Schreibweisen (Syntax) der jeweiligen Programmiersprache einhalten. In der Regel wird eine logische Abfrage mit dem Schlüsselwort **if** eingeleitet, gefolgt von der Abfrageüberprüfung, der **Bedingung**. Danach wird ein **Anweisungsblock** eingefügt. Dieser Anweisungsblock wird *nur dann* ausgeführt, *wenn* die Überprüfung der Bedingung stimmt und **true** ergibt (Speicherzelle = 1).

*Wenn* die *Bedingung* (die Abfrageüberprüfung) stimmt (*true*), *dann* führe die entsprechenden *Anweisungen* (den Anweisungsblock) aus.

```
if Bedingung then Anweisungsblock
```

Eine Bedingung kann auf unterschiedliche Arten definiert (programmiert) werden. Es können Variablen(inhalte) mit Werten verglichen werden, Variablen miteinander, Inhalte von Eingabefeldern mit Werten ...

Nehmen wir zum Beispiel die Eingabe der Lösung einer Rechenaufgabe eines Online-Lerntests. Die Eingabe, das Rechenergebnis des Lerner, soll überprüft werden. Wir fragen den Benutzer im Lerntest nach dem Ergebnis der Rechenaufgabe »5 \* 12« und werten die Eingabe (Ergebnis) des Benutzers in einer logischen Abfrage aus.

Wenn das eingegebene *Ergebnis=60* ist, dann bekommt der Lerner einen zusätzlichen Punkt zur bisherigen Punktzahl hinzu. Die Eingabe des Lerner wird in diesem Fall in einer Variablen mit dem beliebigen von uns festgelegten Namen *Ergebnis* gespeichert. Wir hätten diese Variable auch *Eingabefeld\_Inhalt* oder *Lerner\_Ergebnis* nennen können, wichtig ist nur, dass man am Namen erkennen kann, wozu die Variable dient. Zusätzlich verwenden wir eine Variable *Punktzahl*, die zum Beispiel für vorherige Aufgaben im Online-Lerntest schon verwendet wurde und bereits einen Wert mit der aktuellen Punktzahl des Lerner hat.

*Wie ist das Ergebnis der Rechenaufgabe 5 \* 12? Bitte Antwort eingeben\_*

Die Eingabe des Lerner erfolgt zum Beispiel in einem Eingabefeld und wird nach der Eingabe in der Variablen *Ergebnis* gespeichert (ausgelesen) und anschließend ausgewertet.

```
if Ergebnis=60 then Punktzahl=Punktzahl + 1
```

Wenn in der Variablen mit dem Namen *Ergebnis* die Zahl *60* gespeichert ist, dann zähle einen Punkt zur Variablen *Punktzahl* hinzu.

Es ist nicht in jeder Programmiersprache zwingend notwendig, Bedingungen logischer Abfragen in *runde* Klammern zu setzen (einzufassen), aber es erhöht in jedem Fall die Übersicht und vermeidet logische Fehler, besonders bei verschachtelten Bedingungen. Bedingungen werden dann als eine Art »Gruppe« betrachtet. Wir sollten aber Bedingungen in Klammern schreiben, um diese für uns hervorzuheben. Bedingungen in runden Klammern werden in jeder Programmiersprache akzeptiert und bei einigen sogar verlangt.

```
if (Ergebnis=60) then Punktzahl=Punktzahl+1
```

### ► Wie funktioniert eine if/then-Abfrage?

Wir können es uns so vorstellen, dass die Bedingung in der Klammer ausgewertet (überprüft) wird, und das Ergebnis dieser Auswertung entscheidet, ob der Anweisungsblock ausgeführt wird oder nicht. Die Speicherzellen der Variablen *Ergebnis* (der Variableninhalt) werden mit der Dualzahl *60* ( $60_{\text{dezimal}}=00111100_{\text{dual}}$ ) Bit für Bit (bitweise) mit dem logischen *UND*-Operator verglichen. Nur wenn alle Bitpositionen (Dualstellen) gleich sind, gibt uns das Programm sein »OK« und schickt uns *true* (1) als positives Ergebnis zurück.

<b>Benutzereingabe</b>	60	0	0	1	1	1	1	0	0
<b>Vergleichswert</b>	60	0	0	1	1	1	1	0	0
<b>UND Vergleich</b>	1	1	1	1	1	1	1	1	1

Tabelle 3.1: Richtiger Vergleich mit dem logischen Operator UND

Ist dabei nur eine Bitposition unterschiedlich, wird als Ergebnis des Vergleichs false (0) zurückgeschickt (zurückgeliefert). Mit zurückgeschickt ist gemeint, dass das Ergebnis des bitweisen Vergleichs »vorübergehend« in einer Speicherzelle gespeichert wird und die Überprüfung eigentlich nur das Auslesen (Auswerten) dieser Speicherzelle ist. Diese Speicherzelle ist somit eine lokale Variable vom Typ Boolean, die nur für den Vergleich benötigt und erstellt und anschließend wieder vom System gelöscht wird.

<b>Benutzereingabe</b>	62	0	0	1	1	1	1	1	0
<b>Vergleichswert</b>	60	0	0	1	1	1	1	0	0
<b>UND Vergleich</b>	0	1	1	1	1	1	1	0	1

Tabelle 3.2: Falscher Vergleich mit dem logischen Operator UND

Das Ergebnis der Bedingung (des Vergleichs) ist, wie schon beschrieben, **true** oder **false**. Wenn der Benutzer als Antwort 60 eingibt, bekommen wir als Ergebnis **true** und der Anweisungsblock wird ausgeführt. Gibt der Benutzer aber zum Beispiel als Antwort 62 ein, ist das Ergebnis **false**, und der Anweisungsblock wird nicht ausgeführt (übersprungen).

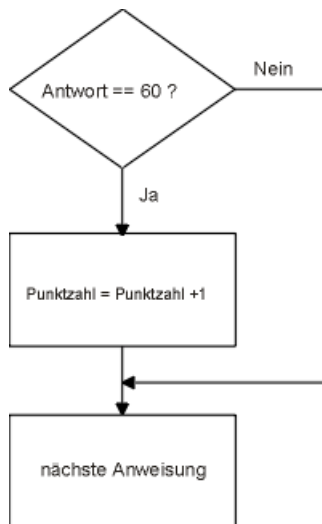


Abbildung 3.2: if/then-Abfrage

Bei der Programmierung kommt es nicht selten vor, dass man logische Abfragen testen muss, ohne die Werte oder Variablen, die überprüft werden sollen, eingeben oder ändern zu können. Wir müssen dann eine Eingabe oder einen Wert »simulieren«, um das Abfrageergebnis vorher beeinflussen oder bestimmen zu können. Soll der Anweisungsblock auf jeden Fall für einen Programmtest ausgeführt werden, können wir »vorübergehend« als Abfragebedingung ( $60=60$ ) schreiben oder (*Ergebnis=Ergebnis*); in diesem Fall ist das Ergebnis natürlich immer **true**. Soll aber der Anweisungsblock auf keinen Fall ausgeführt werden, kann man als Bedingung ( $60=0$ ) oder (*"A"="B"*) schreiben; hier ist das Ergebnis natürlich immer **false**.

Ein Sonderfall logischer Abfragen ist die Überprüfung des Variablentyps Boolean. Diese haben als (Variablen)inhalt *nur* true (1) oder false (0). Zum Beispiel für einen Soundschalter, bei dem der Benutzer einen Sound ein- oder ausschalten kann. Wir definieren eine Booleanvariable und nennen diese Schalterzustand. Diese Variable müssten wir dann normalerweise folgendermaßen für den Schalterzustand »ein« überprüfen

```
if (Schalterzustand=true) then ...
```

oder

```
if (Schalterzustand=1) then ...
```

Da aber das Ergebnis der Abfragebedingung schon ein Booleanwert ist, können wir die Bedingung einer if/then-Abfrage für diesen Variablentyp auch in verkürzter Schreibweise programmieren mit

```
if (Schalterzustand) then ...
```

In der verkürzten Schreibweise wird nicht überprüft, ob das Ergebnis des Vergleichs (*Schalterzustand=true*) *true* ergibt, sondern ob der Inhalt der Variable true ist. Eine Booleanvariable kann mit einer if/then-Abfrage aber auch auf einen falschen Eintrag überprüft werden mit der logischen Abfrage

```
if (Schalterzustand=false) then ...
```

Im Kapitel Schlüsselwörter, Konstanten und Operatoren konnten wir bereits sehen, dass wir mit Hilfe von logischen Operatoren Ergebnisse von Booleanvariablen umkehren können. Wir können so aber auch Ergebnisse von Bedingungen umkehren, da diese einem Booleschen Wert entsprechen. Mit dem logischen Operator **NICHT** (**not, !**) wird die logische Abfrage für die Variable *Schalterzustand* auf das Ergebnis false wie folgt überprüft

```
if not(Schalterzustand) then ...
```

oder

```
if !(Schalterzustand) then ...
```

Wenn die Variable *Schalterzustand* nicht(true) also false ist, dann wird der Anweisungsblock ausgeführt. Auch wenn man sich Anwendungen hierfür im ersten Moment nicht so recht vorstellen kann, ist gerade der logische Operator NICHT ein sehr häufig eingesetztes und sehr nützliches Hilfsmittel.



Wenn wir eine logische Abfrage programmiert haben, das Ergebnis aber später nicht wie erhofft stimmt, können wir gerade den logischen Operator **NICHT** für die Fehlersuche einsetzen. Wir können **not** vor eine Bedingungsklammer schreiben, mit **if not** (Bedingung in der Klammer), um die Abfrage auf falsch zu kontrollieren. Hier sieht man auch, dass es von Vorteil ist, Bedingungen logischer Abfragen in (runde) Klammern zu setzen.

Es kommt aber häufig vor, dass es nicht ausreichend ist, nur das Ergebnis *true* einer Bedingung für das Programm zu verwenden, sondern dass wir *beide* Ergebnisse »*true*« und »*false*« für die Auswertung benötigen. Normalerweise müssten wir alle beiden Ergebnisse in eigenen *if/then*-Abfragen überprüfen. In unserem Beispiel für die Rechenaufgabe könnten wir dies relativ leicht mit zwei *if/then*-Abfragen lösen (programmieren). Ist das Ergebnis richtig, bekommen wir einen Punkt zur Variablen *Punktzahl* hinzu, ist es falsch, bekommen wir einen Punkt abgezogen.

### Die *if/then*-Abfrage für das Abfrageergebnis *true*

```
if (Antwort=60) then Punktzahl=Punktzahl+1
```

Um zu überprüfen, ob eine Antwort *nicht richtig* ist, haben wir mehrere Möglichkeiten. Einmal mit dem logischen Operator *not*(*Antwort=60*) oder indem wir eine Überprüfung programmieren, ob die Antwort *größer oder kleiner* als das Rechenergebnis 60 ist (*Antwort <> 60*).

#### ► Das Abfrageergebnis *false* mit »*not*«

```
if not(Antwort=60) then Punktzahl=Punktzahl-1
```

#### ► Das Abfrageergebnis *false* mit »*<>*«

```
if (Antwort<>60) then Punktzahl=Punktzahl-1
```

Wir können aber bei allen Programmiersprachen auch das Ergebnis »*false*« direkt verwenden. Dazu wird in der Regel das Schlüsselwort **else** (sonst) nach den Anweisungen (Anweisungsblock) für das Ergebnis »*true*« eingefügt und die *if/then*-Abfrage zu einer ***if/then/else***-Abfrage erweitert.

```
if (Antwort=60) then Punktzahl=Punktzahl+1 else Punktzahl=Punktzahl-1
```

Das spart nicht nur Schreibarbeit, sondern wird auch sehr häufig für Lösungen der Programmierung verwendet und benötigt. Wir haben dann einen Anweisungsblock für das Ergebnis »*true*« (true Block) und einen Anweisungsblock für das Ergebnis »*false*« (false Block).




```
if (Bedingung) then
    Anweisungsblock
else
    Anweisungsblock
Abfrage Ende
```

### 3.1.1 Lingo

Macromedia geht mit Lingo mal wieder etwas eigensinnige Wege und unterscheidet sich in der Syntax (Schreibweise) von logischen Abfragen etwas von anderen Programmiersprachen. Das liegt am Ursprung von Director. Lingo war ursprünglich, und ist es teilweise noch immer, eine sehr »bildhafte« Programmiersprache. Lingo wurde für Anwender entwickelt, die keine wirklichen Programmierer waren, wie etwa Designer, oder es »vielleicht« auch nicht werden wollten. Mit Director sollte man sehr leicht und unkompliziert Animationen erstellen können, ohne programmieren zu müssen oder es zu lernen. Erst ab der Version 7 von Macromedia Director wurde Lingo zu einer »wirklich« modernen Programmiersprache komplett überarbeitet und weiterentwickelt. Dies macht Director zu einem sehr starken Werkzeug zur Erstellung von Multimedia-Anwendungen etwa durch die Einführung der Punktnotation oder die Änderung der Listenzugriffe. Kommen wir aber zur Syntax der logischen if/then-Abfragen zurück.

#### Der Syntax der if/then-Abfrage

```
if (Bedingung) then
    Anweisung(en)
end if
```

Die if/then-Abfrage bei Lingo wird mit dem Schlüsselwort **if** eingeleitet, gefolgt von der **Bedingung** in runden Klammern. Die Klammern für die Bedingung sind optional und können weggelassen werden. Nach der Bedingung folgt das Schlüsselwort **then** direkt nach der Bedingung in der *ersten* Zeile und leitet den Anweisungsblock für das Ergebnis *true* ein. Der Anweisungsblock wird mit dem Schlüsselwort **end if** in einer *eigenen neuen* Zeile beendet. Anweisungen *müssen* nach dem Schlüsselwort *then* ebenfalls in *neuen* Zeilen geschrieben werden. Wenn das Ergebnis der Bedingung **true** ist, wird der Anweisungsblock abgearbeitet. Im Falle, dass die Abfrageüberprüfung **false** ergibt, wird der Anweisungsblock aber übersprungen oder ignoriert. Wenn wir eine if/then-Abfrage programmieren wollen, sollte man zuerst die komplette Grundstruktur erstellen, und erst dann die Anweisungen dazwischen einfügen, indem man nach dem Schlüsselwort *then* den Eingabecursor setzt und  (Return, Eingabetaste) drückt. Nachdem man die if/then-Abfrage programmiert hat, kann man nach dem Schlüsselwort *end if* noch einmal  drücken. Das Lingo-Script (der Programmteil) wird dann automatisch ausgerichtet (strukturiert) und rückt in der Regel Anweisungszeilen im Anweisungsblock nach rechts, um diesen hervorzuheben. Dies kann zudem eine gute Kontrolle für uns sein, ob wir richtig programmiert haben. Wenn wir einen Fehler im Quellcode (Programmiersyntax) gemacht haben, werden die Anweisungen *nicht* nach rechts eingerückt. In diesem Beispiel wurde das Schlüsselwort *then* nach  in einer neuen Zeile geschrieben, müsste aber eigentlich in der ersten Zeile der if/then-Abfrage direkt nach der Bedingung stehen.



Anweisungen bei Lingo werden nicht mit einem Semikolon abgeschlossen wie bei JavaScript oder ActionScript.

```

Verhaltensskript 1: Falsche if/then Struktur
Falsche if/then Struktur
1 Intern
global
if (Vorname <> ' ')
then
alert("Geben Sie bitte den Vornamen ein !")
end if

```

Abbildung 3.3: Fehlerhafte if/then-Struktur in Lingo

```

Verhaltensskript 1: Richtige if/then Struktur
Richtige if/then Struktur
1 Intern
global
if (Vorname <> ' ') then
alert("Geben Sie bitte den Vornamen ein !")
end if

```

Abbildung 3.4: Richtige if/then-Struktur in Lingo

Nach der *if/then-Abfrage* wurde  eingefügt und das Script automatisch in einer »Blockstruktur« angeordnet. Gerade bei verschachtelten *if/then-Abfragen* ist uns dies eine sehr nützliche Hilfe, da man sofort erkennen kann, welches Abfrageende mit *end if* zu welcher logischen Abfrage gehört. Wenn eine *if/then-Abfrage* in Lingo nur eine Anweisung hat, können wir die Syntax der Programmierung in einer kürzeren Schreibweise programmieren.

#### ► Verkürzte Syntax der if/then-Abfrage

```
if (Bedingung) then Anweisung
```

In diesem Fall *muss eine* Anweisung direkt hinter dem Schlüsselwort *then* in der gleichen Zeile geschrieben werden (folgen). Die Anweisung könnte aber auch ein Funktionsaufruf sein, bei dem natürlich wiederum mehrere Anweisungen ausgeführt werden können.

Wir können natürlich auch bei Lingo das Ergebnis »false« der Bedingung verwenden, indem wir unsere logische Abfrage mit `else` etwas erweitern.

## Die Syntax der if/then/else-Abfrage

```
if (Bedingung) then
  Anweisung(en)
else
  Anweisung(en)
end if
```

Der erste Teil der if/then/else-Abfrage ist mit der if/then-Abfrage identisch, nur dass der Anweisungsblock für das Ergebnis der Bedingung »true« nicht mit dem Schlüsselwort *end if* abgeschlossen wird, sondern nun das Schlüsselwort *else* folgt. Nach dem Schlüsselwort *else* fügen wir den Anweisungsblock für das Ergebnis »false« ein und beenden danach die if/then/else-Abfrage wie gewohnt mit *end if*. Wir haben sozusagen einen **true-Block** und einen **false-Block** für die if/then/else-Abfrage. Eine if/then/else-Abfrage könnten wir aber auch mit zwei einfachen if/then-Abfragen realisieren.

```
if (Bedingung = true) then
  Anweisung(en)
end if
if (Bedingung = false) then
  Anweisung(en)
end if
```

*Wie man programmiert, bleibt jedem selbst überlassen, da es hierfür keine wirkliche »Ideallösung« gibt. Wobei die Verwendung einer vereinfachten Syntax, oder der zur Verfügung stehenden Methoden einer Programmiersprache zu empfehlen ist.*



## Verschachtelte Bedingungen

Bedingungen logischer Abfragen können auch aus mehreren Bedingungen bestehen und zu einer Bedingung verschachtelt werden, weil zum Beispiel der Anweisungsblock für mehrere Bedingungen gleich ist. Wenn zum Beispiel bei einer Kennworteingabe verschiedene Schreibweisen erlaubt sind, das Kennwort »Aladin« groß geschrieben oder »aladin« klein geschrieben. Wir müssten hierfür normalerweise zwei logische if/then-Abfragen programmieren, die beide den gleichen Anweisungsblock haben.

```
if (Kennwort = "Aladin") then
  Anweisung(en)
end if
if (Kennwort = "aladin") then
  Anweisung(en)
end if
```

Was wäre aber eine Programmiersprache, wenn es nicht auch für dieses »Problem« eine Lösung geben würde, da solche Abfragen zudem auch sehr häufig vorkommen. Die Bedingungen bei der if/then-Abfrage werden mit **logischen Operatoren** zu einer verschachtelten Bedingung mit *einem* Anweisungsblock zusammengefügt.

Wie müssen wir aber Bedingungen verknüpfen, die einen gemeinsamen Anweisungsblock haben? Dazu stellen wir uns in Gedanken selbst einmal die Bedingung für die benötigte if/then-Abfrage selbst. *Wenn* das Kennwort »Aladin« ist *oder* das Kennwort »aladin« ist, *dann* führe den Anweisungsblock aus. Wir sehen auch hier wieder direkt die Lösung. Der logische Operator ODER. Unsere if/then-Abfrage sieht dann wie folgt aus:

```
if (Kennwort="Aladin") or (Kennwort="aladin") then
  Anweisung(en)
end if
```

Wenn wir mit dem logischen Operator `or` zwei Bedingungen in einer if/then-Abfrage verknüpfen, wird der Anweisungsblock ausgeführt, wenn eine der beiden Bedingungen oder alle beide Bedingungen das Ergebnis »true« ergeben. Hier erkennt man auch wieder, dass es von Vorteil ist, Bedingungen in Klammern zu setzen, da zunächst die Bedingungen in den Klammern ausgewertet werden und dann erst der logische Operator die beiden Ergebnisse der Bedingungen vergleicht. Noch besser (sicherer) wäre es, die beiden Bedingungen

```
if ((Kennwort="Aladin") or (Kennwort="aladin")) then ...
```

noch einmal in eine Klammer als Abfragebedingung zu setzen. Der Vergleich von Bedingungen mit dem logischen Operator **ODER** verknüpft ergibt folgende Ergebnisse:

Ergebnis der Bedingung 1	Logischer Operator	Ergebnis der Bedingung 2	Ergebnis der if/then-Bedingung
true	or	true	true
true	or	false	true
false	or	true	true
false	or	false	false

Tabelle 3.3: Ergebnisse der logischen ODER-Verknüpfung

Ein weiterer wichtiger logischer Operator ist das logische **UND**, bei dem alle Bedingungen erfüllt sein *müssen*. Wenn man zum Beispiel ein Kontaktformular überprüfen möchte, bei dem der Benutzer alle Eingaben machen soll, damit die Anweisungen ausgeführt werden. Nur wenn das Eingabefeld für den Nachnamen *und* das Eingabefeld für den Vornamen ausgefüllt wurde, soll das Kontaktformular abgeschickt werden.

In einem Eingabefeld soll hier also überprüft werden, ob ein Eintrag vorgenommen wurde. Ein Eintrag in einem Eingabefeld ist immer vom Typ **String** (Zeichenkette), und wenn ein Eingabefeld leer ist, bedeutet das nicht, dass kein String oder kein Eintrag im Eingabefeld ist, sondern dass der String ein »Leerstring« oder »Nullstring« ist. Ein Nullstring wird bei Lingo und allen anderen Programmiersprachen als String mit Anführungszeichen »""« dargestellt. Mit der Ausnahme, dass zwi-

schen den Anführungszeichen kein Zeichen (Inhalt) für den String steht und dieser somit »leer« ist. Die Überprüfung des Eingabefeldes kann so programmiert werden, dass wir in der Bedingung nicht überprüfen, ob das Eingabefeld leer ist, sondern ob der Eintrag im Eingabefeld *kein* Leerstring ist und somit etwas eingetragen wurde, da die Eingabefelder ursprünglich leer waren.

```
if (Eintrag <> "") then ...
```

oder mit dem logischen Operator

```
if not(Eintrag = "") then ...
```

Wir gehen davon aus, dass der Inhalt vom Eingabefeld Nachname in einer (String)Variablen mit der Bezeichnung Nachname und der Inhalt des Eingabefeldes Vorname in einer Variablen mit der Bezeichnung Vorname gespeichert wurde. Das ergibt folgende verschachtelte if/then-Bedingung:

```
if ((Nachname <> "") and (Vorname <> "")) then
  Anweisung(en)
end if
```

oder mit dem logischen Operator »not«

```
if (not(Nachname = "") and not(Vorname = "")) then
  Anweisung(en)
end if
```

Hier werden zuerst die »inneren« Bedingungen überprüft. Wenn der Nachname eingetragen wurde, wird die Bedingung (*Nachname = ""*) zu **false** und durch den logischen Operator *not* zu **true** umgekehrt. Wenn auch der Vorname eingetragen wurde, wird die Bedingung (*Vorname = ""*) zu **false** und ebenfalls durch den logischen Operator *not* zu **true**. Da beide Ergebnisse »true« ergeben und durch den logischen Operator *and* zwischen den Bedingungen verglichen werden, bekommen wir als endgültiges Ergebnis ebenfalls **true** und die Anweisung wird ausgeführt.

Ergebnis der Bedingung 1	Logischer Operator	Ergebnis der Bedingung 2	Ergebnis der if/then-Bedingung
true	and	true	true
true	and	false	false
false	and	true	false
false	and	false	false

Tabelle 3.4: Ergebnisse der logischen Verknüpfung UND

### 3.1.2 JavaScript, ActionScript

Logische if/then- und if/then/else-Abfragen sind bei JavaScript und ActionScript identisch, da beide Scriptsprachen auch auf der gleichen Norm für Scriptsprachen (ECMA-262) basieren. Die Art und Weise der Programmierung allerdings, die Pro-

grammierlogik einer *if/then*-Abfrage bei JavaScript oder ActionScript, ist wiederum mit der in Lingo vergleichbar, so dass wer sich den Abschnitt für Lingo angesehen hat, sich nur noch um die geänderte Syntax (Schreibweise) kümmern muss.

### Die Syntax der *if/then*-Abfrage

```
if (Bedingung) {Anweisung(en);}
```

Auch hier wird die *if/then*-Abfrage mit dem Schlüsselwort **if** eingeleitet, gefolgt von der **Bedingung** in runden Klammern. Bei JavaScript oder ActionScript wird aber auf das Schlüsselwort *then* verzichtet, da man sowieso davon ausgehen kann, dass das Schlüsselwort *then* folgt.



*Wenn man überall etwas einfügt, um etwas zu beschreiben, damit dies überall gilt, kann man es doch auch gleich weglassen.*

Statt dem Schlüsselwort *then* folgt nach der Bedingung eine geschweifte Klammer »{«, die den Anweisungsblock für das Ergebnis **true** der Abfrageüberprüfung einleitet. Ist das Ergebnis der Bedingung **false**, wird der Anweisungsblock übersprungen. Anweisungen werden bei ActionScript und JavaScript immer mit einem Semikolon »;« beendet, die Ausnahme bilden Anweisungen, die Blöcke einleiten. Der Anweisungsblock wird wieder mit einer geschweiften Klammer »}« beendet (geschlossen), die das Ende der logischen Abfrage anzeigt, wie bei Lingo mit *end if*.



*Jede Art von Blöcken, auch für Funktionen, werden bei JavaScript oder ActionScript in geschweiften Klammern eingefasst.*

Im Gegensatz zu Lingo brauchen wir uns bei JavaScript und ActionScript allerdings nicht um eine Programmstruktur zu kümmern. Die Struktur ist bei diesen beiden Scriptsprachen vollkommen frei. Wir können alles in einer Zeile schreiben, aber wir können auch mehrere Zeilen frei benutzen. Das ist unter anderem der Grund, warum eine Anweisung mit einem Semikolon abgeschlossen werden muss. Der Interpreter geht den Quellcode Zeichen für Zeichen durch und ein Semikolon (markiert) zeigt eine Anweisung als abgeschlossen (beendet) an. Vollkommen egal, ob das Semikolon erst in der nächsten Zeile geschrieben wird. Es empfiehlt sich aber in jedem Fall, eine übersichtliche Struktur bei der Programmierung zu verwenden, um eine bessere Kontrolle sowie Übersicht über das Script zu bekommen. Man kann so im Vorfeld schon Fehler vermeiden. Eine zu empfehlende Struktur sieht etwa so aus:

```
if (Bedingung)
{
    Anweisung(en);
}
```

Das Schlüsselwort *if* schreibt man mit der Bedingung in runden Klammern in die einleitende erste Zeile. Danach schreibt man die geschweifte Klammer »{« in einer neuen Zeile unter das Schlüsselwort *if* und anschließend die Anweisung(en). Wir rücken die Anweisungen etwas nach rechts ein und verwenden für jede Anweisung

eine eigene Zeile mit abschließendem Semikolon. Die Anweisungen bilden somit den Anweisungsblock (true), gefolgt von der geschweiften Klammer »}«, die den Anweisungsblock abschließt. Wenn man wissen will, welche Anweisungen zu welcher if/then-Abfrage gehören, braucht man nur in einer vertikale Linie unter *if* nach unten zu gehen und erkennt dies sehr schnell. Besonders nützlich ist dies bei verschachtelten logischen Abfragen in JavaScript und ActionScript. Solche Blockstrukturen verwendet man auch bei Schleifen oder Funktionen.

#### ► Beispiel

```
if (Bedingung)
{
  Anweisung(en);
  if (Bedingung)
  {
    Anweisung(en);
  }
}
```

### Die Syntax der if/then/else-Abfrage

```
if (Bedingung) {Anweisung;} else {Anweisung;}
```

Schreiben wir die Syntax einer if/then/else-Abfrage direkt in eine etwas übersichtlichere strukturierte Form, um uns von Anfang an einen guten Programmierstil anzugewöhnen. Jeder Programmierer hat seinen eigenen Stil, »seine eigene Handschrift«, aber gewisse Strukturregeln sollte jeder beachten, um es sich und anderen einfacher zu machen.

```
if (Bedingung)
{
  Anweisung(en);
}
else
{
  Anweisung(en);
}
```

Es kommt im Gegensatz zur if/then-Abfrage bei der **if/then/else**-Abfrage nur eine weiterer Block für das Ergebnis »false« der Bedingung in geschweiften Klammern hinzu, gefolgt vom Schlüsselwort **else**. Wie bei Lingo ist der erste Block nach der Bedingung der **true-Block**, der ausgeführt wird, wenn die Bedingung »true« ergibt. Ergibt die Bedingung »false«, wird der **false-Block** nach dem Schlüsselwort *else* ausgeführt.

## 3.2 Die case/switch-Abfrage

Häufig kommt es vor, dass mehrere Varianten einer if/then-Abfrage den gleichen Anweisungsblock oder die gleiche Abfragevariable für verschiedene Werte haben.