Übungen zu Informatik I
Wintersemester 03/04

Übungsleiter: Dipl.-Inform. Tom Gelhausen

Administratives zur Klausur: Anmeldung

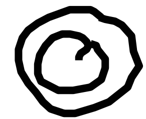
- **Änderung**: Anmeldung der **Physiker**
  - persönliches Erscheinen mit Studentenausweis im Sekretariat erforderlich
  - keine grünen/blauen/rote/lila/… Zettel
  - Adresse: Katja Weißhaupt
AVG, Geb. 50.41
2. OG Raum 203
  - Sprechzeiten: Mo.-Do. 10.00-16.00 Uhr
- **Änderung**: Anmeldung der **Schüler**
  - persönliches Erscheinen mit Personalausweis im Sekretariat erforderlich
  - Adresse und Sprechzeiten wie oben
- **Hinweis**: Die Hörsaalbelegung wird im Web auf http://www.infoeins.de und am schwarzen Brett im AVG, 2. OG (neben dem Briefkasten für die Anmeldezettel) bekannt gemacht.
- **Tipp**: Stellen Sie **rechtzeitig** sicher, dass Sie auf dieser Liste stehen!

Administratives zur Klausur: Abmeldung

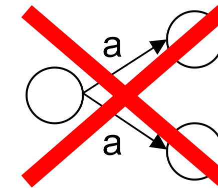
- Bei einer schriftlichen Klausur ist die Abmeldung grundsätzlich bis zum Austeilen der Klausur ohne Attest möglich.
- Wer nicht zur Klausur erscheint muss ein Attest nachliefern
- Bei mündlichen Nachprüfungen ist ein Abmelden ohne Attest nur bis 3 Werktage vor dem Prüfungstermin möglich, danach ist ein Attest erforderlich
- Wenn es Ihnen möglich ist: melden Sie sich bitte bis ca. **4 Tage vor dem Klausurtermin** ab (erleichtert uns die Administration)
- Abmelden: Katja Weißhaupt
AVG, Geb. 50.41
2. OG Raum 203
Sprechzeiten: Mo.-Do. 10.00-16.00 Uhr

Nochmal Automaten…

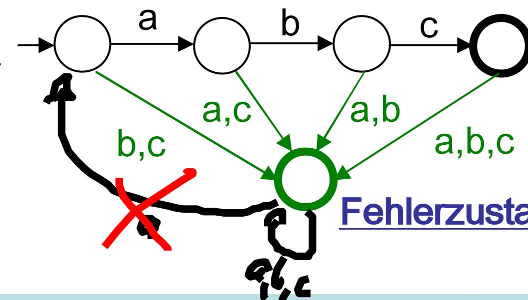
- Formal kann ein endlicher Automat als 5-tupel $\{Z, z_0, E, \Sigma, \delta\}$ definiert werden wobei:
  - Z eine endliche Menge von Zuständen
  - $z_0 \in Z$ der Startzustand
  - $E \subseteq Z$ eine (endliche) Menge möglicher akzeptierender Zustände (auch „Endzustände“, der Automat hält an, wenn er einen akzeptierenden Zustand erreicht)
  - Σ das Eingabealphabet
  - und einer Übergangsfunktion $\delta : Z \times \Sigma \rightarrow Z$



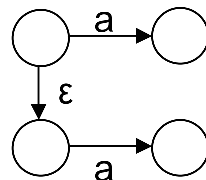
- Vollständiger, deterministischer, endlicher Automat
  - δ ist eine echte Funktion $\Rightarrow$ EA ist deterministisch



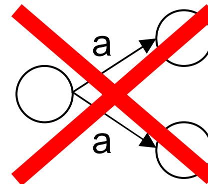
  - δ ist eine totale Funktion $\Rightarrow$ EA ist vollständig



  - $\varepsilon \notin \Sigma$



$\triangleq$



Nochmal reguläre Ausdrücke…

- Darstellung der regulären Ausdrücke als Sprache (Mengenschreibweise)

Regulärer Ausdruck	Beschriebene Sprache	
$\emptyset$	$\mathcal{L}(\emptyset) = \{\}$	I.A.
ε	$\mathcal{L}(\varepsilon) = \{\varepsilon\}$	I.A.
c $(\forall c \in \Sigma)$	$\mathcal{L}(c) = \{c\}$ $(\forall c \in \Sigma)$	I.A.
Seien r, s bereits reguläre Ausdrücke, die die Sprachen R bzw. S beschreiben:		I.V.
(r+s)	$\mathcal{L}(r+s) = R \cup S$	I.S.
rs	$\mathcal{L}(rs) = R \circ S = \{w_1w_2 \mid w_1 \in R \text{ und } w_2 \in S\}$	I.S.
r*	$\mathcal{L}(r^*) = R^* = \{w_1 \ldots w_n \mid w_i \in R \text{ und } n \geq 0\}$	I.S.

Darstellung der regulären Ausdrücke als Sprache

- Seien S und T die Sprachen zu s bzw. t:
  Für welche Sprachen S und T gilt $\mathcal{L}((s+t)^*) = \mathcal{L}((st)^*)$?

⇒ Da es sich bei den Sprachen S und T um Mengen (von Worten) handelt, dürfen wir die Gesetze der Mengenalgebra verwenden, um Aussagen zu beweisen.

- „$\mathcal{L}((s+t)^*) \supseteq \mathcal{L}((st)^*)$": $\mathcal{L}(st) \subseteq \mathcal{L}((s+t)^*)$ gilt.
  ⇒ $\mathcal{L}((st)^*) \subseteq \mathcal{L}(((s+t)^*)^*) = \mathcal{L}((s+t)^*)$
  ⇒ keine Anforderungen an S und T.
- „$\mathcal{L}((s+t)^*) \subseteq \mathcal{L}((st)^*)$": Es gelte $\mathcal{L}(s) \subseteq \mathcal{L}((st)^*)$ und $\mathcal{L}(t) \subseteq \mathcal{L}((st)^*)$
  - Hinreichend:
    ⇒ $\mathcal{L}(s+t) \subseteq \mathcal{L}((st)^*+(st)^*)$
    Es gilt: $\mathcal{L}((st)^*+(st)^*) = \mathcal{L}((st)^*)$
    ⇒ $\mathcal{L}((s+t)^*) \subseteq \mathcal{L}(((st)^*)^*)$
    Es gilt: $\mathcal{L}(((st)^*)^*) = \mathcal{L}((st)^*)$

Darstellung der regulären Ausdrücke als Sprache

Die Bedingung ist also hinreichend. Ist sie auch notwendig?

- Notwendig:

  Sei o.B.d.A. $\mathcal{L}(s) \nsubseteq \mathcal{L}((st)^*)$

  Es gilt $\mathcal{L}(s) \subseteq \mathcal{L}(s+t)$

  und $\mathcal{L}(s+t) \subseteq \mathcal{L}((s+t)^*)$

  $\Rightarrow \mathcal{L}((s+t)^*) \nsubseteq \mathcal{L}((st)^*)$

  Wiederspricht $\mathcal{L}((s+t)^*) \subseteq \mathcal{L}((st)^*)$!

$\Rightarrow$ Anforderungen an S und T: $S \subseteq \mathcal{L}((st)^*)$ und $T \subseteq \mathcal{L}((st)^*)$

Module in Haskell

- In Haskell werden Programmkomponenten auf Module verteilt
- Ein Modul definiert eine Menge von Werten, Datentypen, Klassen, etc.
- Ein Modul exportiert keine, eine, … oder alle der enthaltenen Entitäten
- Module können selbst die Entitäten anderer Module importieren
- Module können (!) importierte Entitäten wieder exportieren
- Jedes Modul ist in einer eigenen Datei mit dem Namen *modulname*∘„.hs“, am besten im gleichen Verzeichnis

Module benutzen

Benutzung:

Zur Verwendung der in einem Modul definierten Datentypen und Funktionen, müssen diese importiert werden.

Mehrere Varianten:

- Komplettes Importieren: `import File`
  importiert alle sichtbaren Typen, Konstruktoren und Funktionen
- Partielles Importieren: `import File (Datei, put, eof)`
  nur die in Klammern angegebenen sichtbaren Typen, Konstruktoren und Funktionen werden importiert
- Qualifiziertes Importieren: `import qualified File`
  importiert alle sichtbaren Typen, Konstruktoren und Funktionen, erlaubt aber die Benutzung nur mit Qualifikation,
  Beispiel: `File.Datei, File.skip`
- Qualifiziertes und partielles Importieren können kombiniert werden.

Module benutzen: Beispiele

Gegeben: Modul `A`, das `x` und `y` exportiert

Import-Anweisung	Sichtbar
`import A`	`x, y, A.x, A.y`
`import A()`	(nichts)
`import A(x)`	`x, A.x`
`import qualified A`	`A.x, A.y`
`import qualified A()`	(nichts)
`import qualified A(x)`	`A.x`
`import A hiding ()`	`x, y, A.x, A.y`
`import A hiding (x)`	`y, A.y`
`import qualified A hiding ()`	`A.x, A.y`
`import qualified A hiding (x)`	`A.y`
`import A as B`	`x, y, B.x, B.y`
`import A as B(x)`	`x, B.x`
`import qualified A as B`	`B.x, B.y`

Exportieren von importierten Entitäten


- ```
  module Ant where

  data Ants = …
  anteater x = …
  ```

- ```
  module Bee where

  import Ant

  beeKeeper = …
  ```

- ```
  module Cow where

  import Bee
  ```

Definitionen von `Ants` und `anteater` in `Cow` nicht sichtbar!

  ```

# Kontrolle über den Export von Entitäten



- `module Bee ( beeKeeper, Ants(..), anteater ) where …`

| | |
|---|---|
| „mit“: | alle Konstruktoren des Typs werden mitexportiert |
| „ohne“: | der Typ wird als abstrakter Datentyp exportiert |

- `module Bee ( beeKeeper, module Ant ) where …`

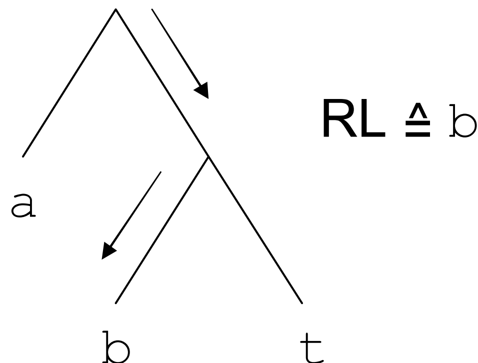
  ⇒ alle Entitäten, die das Modul Ant exportiert, werden auch vom Modul Bee exportiert

- `module` *`modulename`* ist eine Abkürzung für die Auflistung der Namen aller Entitäten

  ⇒ `module Fish where …` ≡ `module Fish ( module Fish ) where …`

# Fallstudie: Huffmancodes

- Erinnerung:
  - Gegeben Zeichenvorrat $\Sigma=\{z_1,\ldots,z_n\}$ und Wahrscheinlichkeiten des Auftretens $p(z_i)=p_i$ $\Sigma p = 1$
  - Ordne Zeichen nach fallender Wahrscheinlichkeit an: $p_1 \geq p_2 \geq \ldots \geq p_n$
  - Fasse die Zeichen mit den niedrigsten Wahrscheinlichkeiten zu neuem Zeichen zusammen und weise diesem Zeichen die Summe der entsprechenden Wahrscheinlichkeiten zu
  - Wiederhole, bis nur ein Zeichen mit p=1 übrig ist
  - Codebaum:




RL ≙ b

Präfixcode: Kein Wort des Codes ist Anfang eines anderen Codes

# Übersicht über die Modulstruktur



Main

decodeMessage
codeMessage

codeTable
codes

MakeCode

Coding

codeTable

makeTree

frequency

CodeTable

MakeTree

Frequency

Types

# Modul Types



```
module Types (Tree(Leaf,Node), Bit(L,R), HCode, Table)
   where

-- Grundtypen für Bits und Huffman-Codes
data Bit = L | R deriving (Eq,Show)
type HCode = [Bit]

-- Codetabelle für's leichtere Codieren/Decodiren
type Table = [(Char,HCode)]

-- Codebaum: Baum zur Darstellung der Häufigkeiten der
-- einzelnen Zeichen und damit der Huffman-Codes.
data Tree = Leaf Char Int | Node Int Tree Tree deriving Show
```

Modul Coding


```
module Coding ( codeMessage , decodeMessage ) where

import Types ( Tree(Leaf,Node), Bit(L,R), HCode, Table )


-- Codiert einen Text nach einer gegebenen Code-Tabelle
codeMessage :: Table -> String -> HCode

-- Decodiert einen Huffman-Code mittels eines gegebenen
-- Huffman-Baumes
decodeMessage :: Tree -> HCode -> String
```

Implementierung:
→ Übungsblatt

Huffmanbäume aufbauen

- Gegeben: „`battat`“
- Frage: Wie kommen wir zu einem optimalen Code?
  - Zuerst müssen wir die Häufigkeiten der einzelnen Buchstaben bestimmen, im **Beispiel**: `[('b',1),('a',2),('t',3)]`
  - Wir verwandeln jetzt jedes Tupel in ein „Bäumchen“, so dass wir eine Liste von Bäumen erhalten:
    `[ Leaf 'b' 1 , Leaf 'a' 2 , Leaf 't' 3 ]`
  - Jetzt verschmelzen wir die Bäume schrittweise:
    - entferne die zwei Bäume mit den geringsten Häufigkeiten aus der Liste
    - diese beiden werden zu einem neuen Baum zusammengefasst
    - die Summe der beiden Häufigkeiten wird in der Wurzel des neuen Baumes notiert
    - der Baum wird an die entsprechende Stelle in die Liste eingefügt
  - … so lange bis es nur noch einen Baum in der Liste gibt!

Verschmelzen der Bäume, in Codetabelle umwandeln

- Ausgangspunkt:
  `[ Leaf 'b' 1 , Leaf 'a' 2 , Leaf 't' 3 ]`
- Entfernen:
  `Leaf 'b' 1` und `Leaf 'a' 2`
  übrig bleibt: `[ Leaf 't' 3 ]`
- Neuer Baum:
  `Node` ??? `(Leaf 'b' 1) (Leaf 'a' 2)`
- Summe einfügen:
  `Node 3 (Leaf 'b' 1) (Leaf 'a' 2)`
- Baum in die Liste einfügen:
  `[Node 3 (Leaf 'b' 1) (Leaf 'a' 2) , Leaf 't' 3 ]`

- … bis nur noch ein einziger Baum übrig bleibt:
  `Node 6 (Node 3 (Leaf 'b' 1) (Leaf 'a' 2)) (Leaf 't' 3)`
- „nur“ noch in die Codetabelle umwandeln:
  `[ ('b',[L,L]) , ('a',[L,R]) , ('t',[R]) ]`

- Neue Datenbank
  - http://db.info.uni-karlsruhe.de
  - Login: Matrikelnummer
  - Passwort: Nachname (bitte ändern)
  - Probleme? → Tutor!

- Jeder hat seine eigene Datenbank
  - Datenbankname: Matrikelnummer (nur da drin arbeiten!!!)
  - Rechte:
    - Tabellen anlegen, ändern, löschen
    - Datensätze anlegen, lesen, ändern, löschen

Datenbank installieren (oder reparieren)

- Die Datei http://www.infoeins.de/Programme/DB.zip
  - herunterladen
  - entpacken
  - im Texteditor öffnen
  - alles markieren und kopieren
  - in die Webseite einloggen
  - Auf den Befehl „SQL" klicken um das entsprechende Eingabefenster zu öffnen
  - alles einfügen
  - auf „Go" klicken
  - abwarten
  - Die Meldung „SQL executed." erscheint. (Nein? → Tutor!)
- Wenn die Datenbank bereits installiert war, müssen zuerst alle Tabellen gelöscht werden („Drop"), bevor obige Prozedur wiederholt werden kann!

SQL – Verschachtelte Anfragen, `in` und `any`

- Schachtelung mittels `any`-Klausel:
  - „Θ **any**“ Qualifiziert jedes äußere Tupel, für das der Vergleich mit irgendeinem Tupel der in der Schachtel berechneten Relation positiv ausfällt
  - **Beispiel**: „Kategorien, von denen mindestens ein Artikel existiert“

```
select *
from   Kategorien
where  KatNr = any (
                 select KatNr
                 from   Artikel
               );
```

  - Abkürzung „`in`“ für „`=any`“:

```
select *
from   Kategorien
where  KatNr in ( select KatNr from Artikel );
```

SQL – Verschachtelte Anfragen, `in` und `all`

- Schachtelung mittels `all`-Klausel:
  - „Θ **all**“ Qualifiziert jedes äußere Tupel, für das der Vergleich mit allen Tupeln der in der Schachtel berechneten Relation positiv ausfällt
  - **Beispiel**: „Kategorien, von denen kein Artikel existiert“

```
select *
from   Kategorien
where  KatNr != all (
                      select KatNr
                      from   Artikel
                    );
```

  - Abkürzung „`not in`“ für „`!=all`“:

```
select *
from   Kategorien
where  KatNr not in ( select KatNr from Artikel );
```

SQL – Verschachtelte Anfragen, `exists`

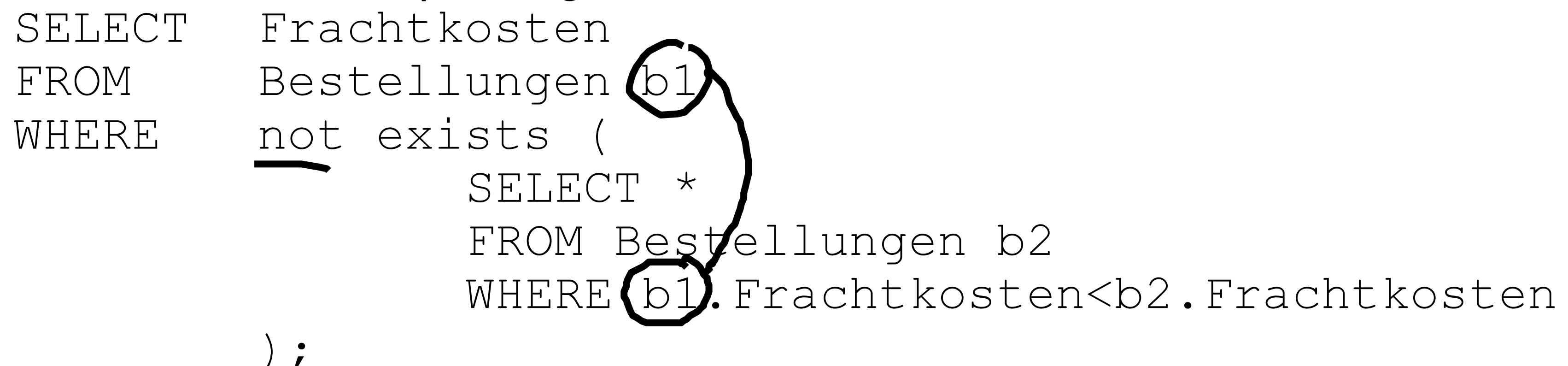
- „Gib die höchsten Frachtkosten aus, die jemals bezahlt wurden. Die Funktion max() darf nicht verwendet werden!“
- Einfach:

```
SELECT max(Frachtkosten)
FROM   Bestellungen;
```

- mit Θ all:

```
SELECT Frachtkosten
FROM   Bestellungen
WHERE  Frachtkosten >= all (
            SELECT Frachtkosten
            FROM Bestellungen
       );
```

- Alternativ Existenzprüfung mit „`exists`“:



```
SELECT   Frachtkosten
FROM     Bestellungen b1
WHERE    not exists (
                SELECT *
                FROM Bestellungen b2
                WHERE b1.Frachtkosten<b2.Frachtkosten
         );
```

Existenzquantifizierung – explizit und implizit


```
SELECT  Frachtkosten
FROM    Bestellungen b1
WHERE   exists (
             SELECT *
             FROM Bestellungen b2
             WHERE b1.Frachtkosten<b2.Frachtkosten
        );
```

- Existenzquantifizierung kann man auch implizit machen:

```
SELECT  DISTINCT b1.Frachtkosten
FROM    Bestellungen b1, Bestellungen b2
WHERE   b1.Frachtkosten < b2.Frachtkosten;
```

SQL – Einfügen von Tupeln

- Syntax:
  `INSERT INTO` *relationsname* [*attributnamensliste*]
  { `VALUES ( ` *wertliste* ` )` | *selectanweisung* }`;`
- Jede Einfügeanweisung bezieht sich stets auf eine einzige Relation *relationsname*.
- Falls nicht alle Tupelwerte belegt werden sollen, können die zu belegenden über die optionale Attributliste angegeben werden. Die restlichen Werte werden auf `NULL` gesetzt (siehe nächste Folie).
- Ohne Attributliste müssen die eingefügten Tupel zum Schema der Relation kompatibel sein
  - kompatible Typen
  - Anzahl
  - Reihenfolge

  } Konsistenzbedingungen beachten!
- Einfügevarianten
  - Einfügen von Einzeltupeln: Verwendung des Schlüsselwortes `VALUES`, gefolgt von einer Wertliste
  - Einfügen von Tupelmengen: Angabe einer SQL-Anfrage

- Es kann sein, dass man den Benutzer nicht zwingen will, einen Wert für eine bestimmte Tupelkomponente einzugeben
- Abhilfe: `NULL`-Wert
  - Theorie: Kapitel 2(a) Folie 69 „flach geordnete Mengen“
  - `null` passt überall: int, float, char, varchar, date, …
  - `null` darf standardmäßig überall verwendet werden, es kann jedoch explizit verboten werden, `null`-Werte einzusetzen, indem man bei der Schemadefinition den Parameter „`NOT NULL`“ setzt.

Field	Type	Not Null	Default	Actions	
ANr	integer	NOT NULL		Alter	Drop
Artikelname	character varying(40)			Alter	Drop
Lieferant	integer			Alter	Drop
KatNr	integer			Alter	Drop

  - Auf den `null`-Wert kann man in prüfen: „`IS [NOT] NULL`“
- **Beispiel**: Person ≔ (name:string, middleinitial:char, givenname:string)

```
INSERT INTO Person values ('Gelhausen',null,'Tom');
SELECT * FROM Person WHERE middleinitial IS NULL;
```

SQL – Einfügen von Tupeln: Beispiele

- **Beispiel**: Einfügen einer neuen Kategorie in die Relation Kategorien:

```
INSERT INTO Kategorien
    VALUES   (9, 'Ersatzteile', 'Zündkerzen, etc.');
```

Reihenfolge! Anzahl! Typen!

- **Beispiel**: Angenommen, wir hätten eine Tabelle, die neue, bisher ungenutzte Kategorien enthält. Der Name dieser Tabelle sei „VorbereiteteKategorien" und habe ein identisches Schema zu Kategorien. In dieser Relation sei der neue Datensatz bereits enthalten:

```
INSERT INTO Kategorien
    SELECT   *
    FROM     VorbereiteteKategorien
    WHERE    KatNr=9;
```

- **Beispiel**: Einfügen unter Zulassen von `NULL`-Werten und automatischem Generieren des Primärschlüssels:

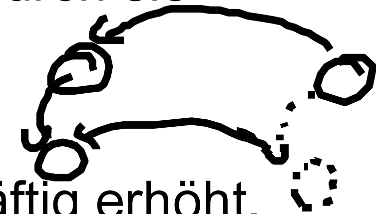
```
INSERT INTO Kategorien ( KatNr, Name )
    SELECT   max(KatNr)+1, 'Ersatzteile'
    FROM     Kategorien;
```

SQL – Ändern von Tupeln

- Syntax:

  `UPDATE` *relationsname*
  `SET` *attributname* `=` *ausdruck* {`,` *attributname* `=` *ausdruck*}*
  [`WHERE` *bedingung*] ;

- Bezieht sich stets auf Mengen (auch 0- und 1-elementige) von Tupeln einer Relation.
- Bei der Verwendung der `WHERE`-Klausel werden nur die durch sie qualifizierten Tupel geändert.



- Konsistenzbedingungen beachten!
- **Beispiel**: Der Lieferant „Lyngbysild“ hat seine Preise kräftig erhöht. Das soll jetzt in der Datenbank eingetragen werden. Weil sich so teure Artikel jedoch nicht verkaufen lassen, sollen sie aus dem Sortiment gestrichen werden.

```
UPDATE  Artikel
SET     Einzelpreis = Einzelpreis*2, Auslaufartikel = 1
WHERE   Lieferant IN
   (SELECT LNr FROM Lieferant WHERE Name='Lyngbysild');
```

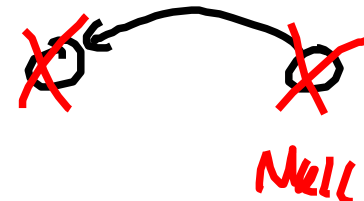
- Syntax

```
DELETE FROM  relationsname
[ WHERE      bedingung ] ;
```

- Entweder werden alle Tupel einer Relation oder, bei Angabe einer WHERE-Klausel, die durch die Suchbedingung selektierten Tupel gelöscht.

- **Beispiel**: „Lyngbysild" hat seine Preise noch mal erhöht. Die Artikel sollen sofort aus dem Sortiment genommen werden

```
DELETE FROM  Artikel
WHERE        Lieferant IN
                (SELECT LNr
                 FROM   Lieferant
                 WHERE  Name='Lyngbysild');
```



- Konsistenzbedingungen beachten!

Übungsblätter

- Sowie die Musterlösungen finden Sie unter

http://www.infoeins.de/uebungsblaetter.php