Einführung in Datenbanken

Kapitel 5: Logik in SQL, Teil I: AND, OR, NOT und Joins

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2019/20

http://www.informatik.uni-halle.de/~brass/db19/

Inhalt

Relationale Datenbanken (1)

- In relationalen Datenbanken werden die Daten in Tabellen abgespeichert, z.B.

STUDENTEN		
SID	VORNAME	NACHNAME
101	Lisa	Weiss
102	Michael	Grau
103	Daniel	Sommer
104	Iris	Winter

Zur Vereinfachung wurde die Spalte "EMAIL" hier weggelassen.

- Statt Zeilen spricht man formal auch von "Tupeln".

Bei einer Tabelle mit drei Spalten wie im Beispiel entsprechen die Tabellenzeilen Tripeln (3-Tupel), z.B. `(101, 'Lisa', 'Weiss')`.

Relationale Datenbanken (2)

- In der Logik kann man den Zugriff auf Tabellenzeilen auf zwei verschiedene Arten formalisieren:
  - Bereichskalkül (BK): Eine Tabelle mit n Spalten entspricht einem n-stelligen Prädikat: $p(t_1, \ldots, t_n)$ ist wahr gdw.

t_1	$\cdots$	t_n

    eine Zeile der Tabelle ist.
  - Tupelkalkül (TK): Eine Tabelle mit n Spalten entspricht einer Sorte mit n Zugriffsfunktionen, die die Werte der Spalten liefern.

    Alternativ statt Sorte auch einstelliges Prädikat, siehe unten.

- SQL basiert auf dem Tupelkalkül,
  Datalog und QBE auf dem Bereichskalkül.

Relationale Datenbanken (3)

- Im Beispiel würde der Bereichskalkül ein Prädikat `STUDENTEN` einführen.
  - `STUDENTEN(101, 'Lisa', 'Weiss')` wäre wahr.
  - `STUDENTEN(200, 'Martin', 'Mueller')` wäre falsch.

    Im Bereichskalkül laufen Variablen über Datentypen (`int`, `string`).

    Natürlich würde man besser Tabellennamen im Singular wählen.

- Der Tupelkalkül würde eine Sorte `STUDENTEN` einführen, sowie Zugriffsfunktionen `SID`, `VORNAME`, `NACHNAME`.
  - Für ein `X` der Sorte `STUDENTEN` gilt dann: `SID(X) = 101`, `VORNAME(X) = 'Lisa'`, und `NACHNAME(X) = 'Weiss'`.

    Im Tupelkalkül laufen Variablen über ganzen Tupeln.

    Man führt dann die alternative Notation `X.SID` für `SID(X)` ein.

Relationale Datenbanken (4)

- Tatsächlich ist die obige Tupelkalkül-Variante schon auf SQL angepasst.
- Im theoretischen Tupelkalkül kann man Variablen über beliebigen Tupeln deklarieren, z.B.

  $\langle$`SID: int, VORNAME: string, NACHNAME: string`$\rangle$

  Dieser ganze Ausdruck ist eine Sorte. Das Alphabet ist ja ohnehin unendlich, insofern ist das kein Problem. Natürlich kann man eine Abkürzung für den Tupel-Typ einer Tabelle einführen, aber Variablen können auch über Tupel-Typen laufen, die in der Datenbank nicht vorkommen (z.B. für das Anfrage-Ergebnis).
- Die Tabellen entsprechen dann einstelligen Prädikaten. Z.B. kann man mit der Bedingung `STUDENTEN(S)` fordern, dass der Wert einer Variablen `S` der Tupelsorte tatsächlich als Zeile der Tabelle vorkommt.

Relationale Datenbanken (5)

- Vorteil von SQL gegenüber theoretischem TK:
  - Es werden keine Einschränkungen auf den Formeln benötigt, um zu garantieren, dass Variablen nur über einem endlichen Bereich laufen.

    Damit die Formeln effektiv auswertbar sind, muss sichergestellt werden, dass es ausreicht, nur endlich viele Werte für jede Variable auszuprobieren. Dafür wurden die Bedingungen "Bereichsunabhängigkeit" und "Bereichsbeschränkung" eingeführt. In SQL kann man keine Formeln aufschreiben, die dagegen verstoßen würden.
- Nachteil von SQL:
  - Für Ergebnis-Spalten, in denen Werte aus verschiedenen Datenbank-Spalten vorkommen, wird ein `UNION`-Operator benötigt. Logische Formeln reichen hier nicht.

Tupelkalkül — SQL-Variante (1)

- Ein DBMS definiert eine Menge von Datentypen (z.B. Strings, Zahlen) mit Konstanten, Datentypfunktionen (z.B. $+$) und Prädikaten (z.B. $<$).
- Für diese definiert das DBMS Namen (in der Signatur $\Sigma_{\mathcal{D}}$) und Bedeutung (in der Interpretation $\mathcal{I}_{\mathcal{D}}$).
- Normalerweise gibt es für jeden Datenwert $d \in \mathcal{I}_{\mathcal{D}}[s]$ mindestens eine Konstante c mit $\mathcal{I}_{\mathcal{D}}[c] = d$.

  D.h. alle Datenwerte sind durch Konstanten benannt. Das wird auch Bereichsabschlußannahme genannt und ist z.B. zur Ausgabe von Datenwerten im Anfrageergebnis wichtig. Im Allgemeinen können verschiedene Konstanten den gleichen Datenwert bezeichnen, z.B. `0`, `00`, `-0`.

Tupelkalkül — SQL-Variante (2)

- Aufgrund der Angaben im DB-Schema wird die Signatur $\Sigma_{\mathcal{D}}$ zur Signatur Σ für Anfragen erweitert, und zwar um:
  - Sorten (eine für jede Relation/Tabelle).

    Während die Interpretation der Datentypen fest in das DBMS eingebaut ist, kann die Interpretation der zusätzlichen Sorten durch Einfügungen, Löschungen und Updates verändert werden.
    Dafür müssen diese Sorten einen endlichen Wertebereich haben.
    Diese Sorten werden ja durch Dateien auf der Platte implementiert, während die Datentypen durch Programmcode im DBMS implementiert sind und daher unendliche Wertebereiche möglich wären, z.B. beliebig lange Zeichenketten.
  - einstellige Funktionen, jeweils von einer der neuen Sorten in einen Datentyp (für jede Spalte).

    Dies sind Zugriffsfunktionen für die Attribute/Komponenten der Zeilen/Tupel/Records/Objekte.

Tupelkalkül — SQL-Variante (3)

- In der Punkte-DB gibt es z.B. die Sorte `STUDENTEN` mit den Funktionen
  - `SID: STUDENTEN` $\rightarrow$ `NUMERIC(3)`
  - `VORNAME: STUDENTEN` $\rightarrow$ `VARCHAR(20)`
  - `NACHNAME: STUDENTEN` $\rightarrow$ `VARCHAR(20)`

    Die Werte der Sorte `STUDENTEN` sind die Tabellenzeilen.
    Jede der Funktionen liefert den Wert der entsprechenden Spalte für die als Argument gegebene Tabellenzeile. Diese Tabelleneinträge sind Werte aus einem Datentyp wie `NUMERIC(3)` oder `VARCHAR(20)`.
    Diese Zugriffsfunktionen wählen also einfach jeweils eine Komponente des Tupels aus. `EMAIL` erlaubt auch Nullwerte, formal erst in Kap. 8.

- Beispiel für Überladung eines Funktionssymbols:
  - `SID: BEWERTUNGEN` $\rightarrow$ `NUMERIC(3)`

Tupelkalkül — SQL-Variante (4)

- Z.B. enthält $\mathcal{I}[\texttt{STUDENTEN}]$ das Tupel

$$t = (101, \text{'Lisa'}, \text{'Weiss'})$$

- Dann ist $\mathcal{I}[\texttt{SID}](t) = 101$.
- Selbstverständlich müssen die neuen Sorten als endliche Mengen interpretiert werden (ggf. auch leer).

  Man fordert auch, dass es keine zwei verschiedenen Tupel geben kann, die in den Werten aller Zugriffsfunktionen übereinstimmen. Dies ist für die Äquivalenz zum Bereichskalkül wichtig, da dort ein Prädikat nicht zweimal wahr sein kann. In der Praxis (SQL) könnte es tatsächlich solche Tupel geben, die in allen Komponenten übereinstimmen. Fast immer werden aber Schlüssel für eine Relation/Tabelle definiert, und dann kann dieser Fall wieder nicht auftreten.

Beispiel-Datenbank

STUDENTEN

SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	···
102	Michael	Grau	NULL
103	Daniel	Sommer	···
104	Iris	Winter	···

AUFGABEN

ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

BEWERTUNGEN

SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
102	H	2	9
102	Z	1	10
103	H	1	5
103	Z	1	7

Inhalt

Variablendeklaration (1)

Definition:

- Sei die Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ gegeben.
- Eine Variablendeklaration für Σ ist eine partielle Abbildung $\nu : \mathit{VARS} \to \mathcal{S}$, so dass
  - der Definitionsbereich von ν endlich ist, und

    Das ist keine Einschränkung, weil jede Formel nur endlich viele Variablen enthält.
  - $\nu(c)$ für $c \in \mathcal{F}_\epsilon \cap \mathit{VARS}$ undefiniert ist.

    Wenn ein Bezeichner schon als Konstante verwendet wird, kann er nicht gleichzeitig als Variable benutzt werden, weil sonst Vorkommen des Bezeichners in Formeln mehrdeutig wären.
- Variablendeklarationen werden auch in der Form $\nu = \{\mathtt{S/STUDENTEN},\ \mathtt{B/BEWERTUNGEN}\}$ geschrieben.

Variablendeklaration (2)

- Variablendeklarationen definieren, welche Variablen verwendet werden können, und was ihre Sorten sind:

ν	
Variable	Sorte
S	STUDENTEN
B	BEWERTUNGEN

  Jede Variable muss eine eindeutige Sorte haben.

- In SQL werden Variablendeklarationen unter `FROM` definiert:

```
FROM STUDENTEN S, BEWERTUNGEN B
```

- Im Tupelkalkül und in SQL kann man Variablen nur für Tupel-Sorten deklarieren (über Tabellenzeilen laufen lassen).

  Nicht für Datensorten (wie `int`). Sichert endliche Auswertbarkeit von SQL.

Tupelvariablen in SQL (1)

- Die `FROM`-Klausel legt eine Variablendeklaration ν fest:

```
SELECT A.ANR, A.THEMA
FROM   AUFGABEN A
WHERE  A.ATYP = 'H'
```

Hier wird eine Variable `A` der Sorte `AUFGABEN` deklariert. Ein Datenbankler würde eher sagen: "Eine Variable `A`, die über den Zeilen der Tabelle `AUFGABEN` läuft". Oder kurz: "eine Variable `A` über `AUFGABEN`". Oder: "eine Variable `A` für `AUFGABEN`".

Variable	Sorte/Tabelle
A	AUFGABEN

- Weil `A` vom Typ `AUFGABEN` ist, können die Zugriffsfunktionen für die Spalten von `AUFGABEN` verwendet werden, z.B. `ANR`.
- In SQL schreibt man `A.ANR` statt `ANR(A)`.

Tupelvariablen in SQL (2)

- Eine Tupelvariable wird immer erstellt: Ist kein Name angegeben, erhält sie den Namen der Relation:

```
SELECT AUFGABEN.ANR, AUFGABEN.THEMA
FROM   AUFGABEN
WHERE  AUFGABEN.ATYP = 'H'
```

Variable	Sorte/Tabelle
AUFGABEN	AUFGABEN

- Wenn man also nur `FROM AUFGABEN` schreibt, wird dies behandelt wie:

```
FROM   AUFGABEN AUFGABEN
```

Die Tupelvariable "`AUFGABEN`" läuft über alle Zeilen der Tabelle "`AUFGABEN`".

Tupelvariablen in SQL (3)

- Wird ein Tupelvariablen-Name angegeben, z.B.

  `FROM AUFGABEN A`

  so ist es ein Fehler, "`AUFGABEN.ANR`" zu schreiben.

  Z.B. Oracle: `ORA-00904: "AUFGABEN"."ANR": invalid identifier.`

  PostgreSQL:

  ```
  invalid reference to FROM-clause entry for table "AUFGABEN".
  HINT: Perhaps you meant to reference the table alias "a".
  ```

  MariaDB:

  ```
  ERROR 1054 (42S22): Unknown column 'AUFGABEN.A' in 'field list'
  ```

- Die Tupelvariable heißt nun "`A`", nicht "`AUFGABEN`".

  Wenn man möchte, kann man die Situation auch so verstehen, dass hier ein "Alias" `A` für die Tabelle `AUFGABEN` eingeführt wurde, und deswegen nur die umbenannte Tabelle `A` zur Verfügung steht. Dies wäre nicht die Sicht der Logik, aber man kann SQL auch aus Sicht der relationalen Algebra verstehen (Kap. 9).

Tupelvariablen in SQL (4)

- Wenn mehrere Tupelvariablen unter `FROM` deklariert werden, ist die Reihenfolge egal:

```
FROM STUDENTEN S, BEWERTUNGEN B
```

und

```
FROM BEWERTUNGEN B, STUDENTEN S
```

führen beide zur gleichen Variablendeklaration.

Mathematisch ist es ja eine (partielle) Abbildung von Variablen auf Sorten.
Es wäre möglich, dass der Anfrageoptimierer einen unterschiedlichen Zugriffsplan wählt, also die eine Anfrage etwas schneller läuft (bei guten Optimierern selten).

- Man darf die gleiche Variable nicht für verschiedene Tabellen verwenden:

```
FROM BEWERTUNGEN X, STUDENTEN X
```

PostgreSQL: "`ERROR: table name "x" specified more than once`".
Die Funktionseigenschaft der Variablendeklaration ist verletzt.

Variablenbelegung

Definition:

- Eine Variablenbelegung $\mathcal{A}$ für $\mathcal{I}$ und ν ist eine partielle Abbildung von $VARS$ auf $\bigcup_{s \in \mathcal{S}} \mathcal{I}[s]$.

  $\mathcal{A}$ wie "Assignment" (kurz für "Variable Assignment").

- Sie definiert für jede Variable V, für die ν definiert ist, einen Wert aus $\mathcal{I}[s]$, wobei $s := \nu(V)$.

  Für Variablen V, die in ν nicht deklariert sind (d.h. ν ist undefiniert für V) ist auch $\mathcal{A}$ undefiniert. Damit ist auch eine Variablenbelegung endlich aufschreibbar, z.B. wie Variablendeklarationen: $\{X_1/d_1, \ldots, X_n/d_n\}$.

Erklärung:

- D.h. eine Variablenbelegung legt für alle Variablen, die in ν deklariert sind, Werte aus $\mathcal{I}$ fest (passender Sorte).

Variablenbelegungen in SQL (1)

Beispiel:

- Es sei folgende Variablendeklaration ν gegeben:

Variable	Sorte
X	STUDENTEN

- Eine mögliche Variablenbelegung ist:

X $\longrightarrow$

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

Mathematisch: $\mathcal{A}(X) = (101, \text{'Lisa'}, \text{'Weiss'}, \text{'...'}) \in \mathcal{I}[\texttt{STUDENTEN}]$

Variablenbelegungen in SQL (2)

- Beispiel:

```
SELECT X.NACHNAME
FROM   STUDENTEN X
WHERE  X.VORNAME = 'Daniel'
```

- Ein simples Auswertungsverfahren ist, eine alle möglichen Variablenbelegungen $\mathcal{A}$ durchzugehen, also `X` nacheinander jede Zeile zuzuweisen:

```
for X in STUDENTEN do
    if X.VORNAME = 'Daniel' then
        print X.NACHNAME
```

`X` steht hier für eine Zeile in `STUDENTEN`. Man testet jeweils, ob die `WHERE`-Bedingung erfüllt ist. Falls ja, wird der Wert des Ausdrucks unter `SELECT` ausgedruckt.

Variablenbelegungen in SQL (3)

- Erste Variablenbelegung:

X ⟶

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

- `X.VORNAME` ist "`Lisa`". Die `WHERE`-Bedingung

  ```
  X.VORNAME = 'Daniel'
  ```

  ist nicht erfüllt.

- Es erfolgt keine Ausgabe für diese Variablenbelegung.

Variablenbelegungen in SQL (4)

- Zweite Variablenbelegung:

X ⟶

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

- Für diese Variablenbelegung hat `X.VORNAME` den Wert "`Michael`".
- Wieder ist die `WHERE`-Bedingung nicht erfüllt.
  Wieder keine Ausgabe.

Variablenbelegungen in SQL (5)

- Dritte Variablenbelegung:

X ⟶

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

- Für diese Variablenbelegung hat `X.VORNAME` den Wert "`Daniel`". Die `WHERE`-Bedingung ist erfüllt.
- Daher wird der Wert des `SELECT`-Ausdrucks `X.NACHNAME` für diese Variablenbelegung ausgegeben. Dies ist "`Sommer`".

Variablenbelegungen in SQL (6)

- Vierte Variablenbelegung (wieder keine Ausgabe):

	STUDENTEN			
	SID	VORNAME	NACHNAME	EMAIL
	101	Lisa	Weiss	...
	102	Michael	Grau	NULL
	103	Daniel	Sommer	...
X⟶	104	Iris	Winter	...

- Ein DBMS muss nur die gleiche (Multi-)Menge von Ausgaben liefern, als wenn es alle möglichen Variablenbelegungen durchprobiert hätte. Es darf cleverere Methoden verwenden.

  Falls es einen Index über der Spalte `VORNAME` der Tabelle `STUDENTEN` gibt, könnte dieser verwendet werden, um schnell Zeilen mit `VORNAME = 'Daniel'` zu finden. Für eine so kleine Tabelle lohnt sich ein Index aber nicht.

Verbunde/Joins (1)

- Gegeben sei eine Anfrage mit zwei Tupelvariablen:

```
SELECT A_1, ..., A_n
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  C
```

- Dann wird `S` über die 4 Tupel in `STUDENTEN` laufen und `B` über die 8 Tupel in `BEWERTUNGEN`. Im Prinzip werden alle $4 * 8 = 32$ Kombinationen betrachtet:

```
for S in STUDENTEN do
    for B in BEWERTUNGEN do
        if C then print A_1, ..., A_n
```

Ein gutes DBMS verwendet evtl. einen besseren Algorithmus zur Auswertung der Anfrage (hängt von Bedingung C ab). Aber um die Bedeutung der Anfrage zu verstehen, reicht der einfache Algorithmus.

Verbunde/Joins (2)

- Interessant sind aber nur Variablenbelegungen, bei denen sich `S` und `B` auf den gleichen Studenten beziehen:

```
SELECT S.NACHNAME, B.ATYP, B.ANR, B.PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
```

- Bedingungen, die Spaltenwerte verschiedener Tupelvariablen vergleichen, werden auch "Verbundbedingungen" (engl. "Join Conditions") genannt.

  Der Verbund ist eine Operation der relationalen Algebra, und wird in Kapitel 9 ausführlich besprochen. Er klebt sozusagen Zeilen zweier Tabellen "zusammen", die die Verbundbedingung erfüllen (hier aus `STUDENTEN` und `BEWERTUNGEN`).

- Typisch ist, dass die Gleichheit von Fremdschlüssel und referenziertem Schlüssel gefordert wird (wie hier).

Verbunde/Joins (3)

- Beispiel für interessante Variablenbelegung (erfüllt Bedingung):

S ⟶

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

B ⟶

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
⋮	⋮	⋮	⋮

Verbunde/Joins (4)

- Auch diese Variablenbelegung produziert eine Ausgabe:

S⟶

STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

B⟶

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
⋮	⋮	⋮	⋮

Verbunde/Joins (5)

- Diese Variablenbelegung dagegen erfüllt die Bedingung nicht:

S⟶

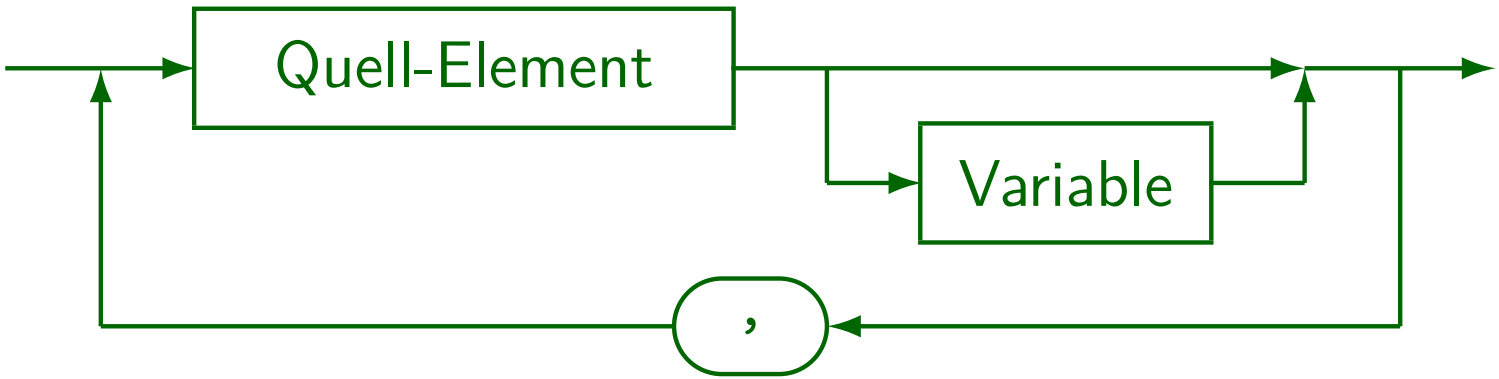
STUDENTEN			
SID	VORNAME	NACHNAME	EMAIL
101	Lisa	Weiss	...
102	Michael	Grau	NULL
103	Daniel	Sommer	...
104	Iris	Winter	...

B⟶

BEWERTUNGEN			
SID	ATYP	ANR	PUNKTE
101	H	1	10
101	H	2	8
101	Z	1	12
102	H	1	9
⋮	⋮	⋮	⋮

FROM-Syntax (1)

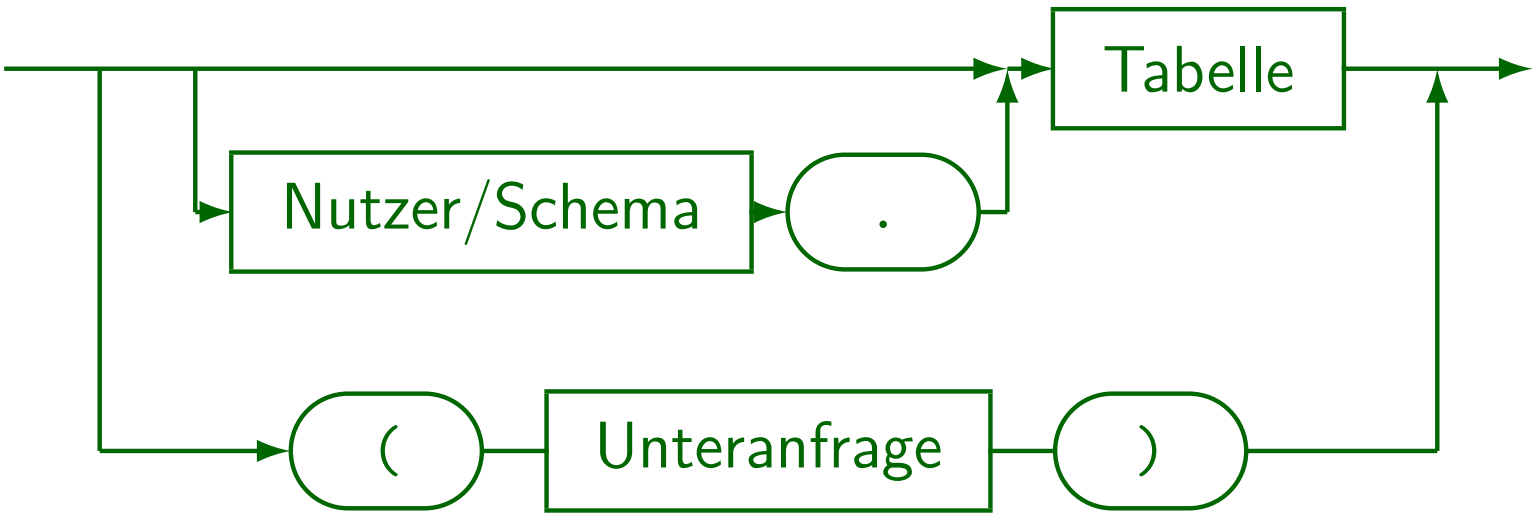
Quell-Liste (nach FROM):



- In SQL-92, SQL Server, Access, DB2 und MySQL (nicht in Oracle 8i) kann man "`AS`" zwischen Quell-Element und Variable schreiben.
- In SQL-92, DB2 (nicht Oracle, SQL Server, Access, MySQL) kann man neue Spaltennamen definieren: "`STUDENTEN AS S(NR,VNAME,NNAME,MAIL)`".
- Ist das "Quell-Element" eine Unteranfrage, wird in SQL-92, SQL Server und DB2 eine Tupelvariable verlangt (aber nicht Oracle, Access). Dann funktioniert obige Spaltenumbenennung plötzlich auch in SQL Server.
- SQL-92, SQL Server, Access, DB2 unterstützen Joins unter `FROM` (später).

FROM-Syntax (2)

Quell-Element:



- SQL-86 erlaubt keine Unteranfragen in der `FROM`-Liste.
- MySQL unterstützt Unteranfragen erst ab Version 4.1.
- Vereinfachte Syntax der `FROM`-Klausel:

```
FROM Tabelle [Variable], ..., Tabelle [Variable]
```

FROM-Syntax (3)

Tabellennamen:

- Man kann sich auf Tabellen anderer Nutzer unter `FROM` beziehen (falls Leserecht erteilt wurde):

```
SELECT * FROM BRASS.AUFGABEN
```

- Der Nutzername ist hier der Name des DB-Schemas (ein DBMS kann mehrere Schemata verwalten).

  In Oracle sind Nutzer und Schema mehr oder weniger das gleiche: Jeder Nutzer hat sein eigenes Schema, jedes Schema gehört genau einem Nutzer. In DB2 kann es mehrere Schemata je Nutzer geben. In PostgreSQL sind Schema und Nutzer getrennt. Eine Datenbank kann mehrere Schemata enthalten, und Nutzer haben einen Suchpfad für Schemata. Man kann aber auch "Schema.Tabelle" schreiben. In SQL Server hat ein vollständiger Name die Form "Server.DB.Inhaber.Tabelle", aber es gibt viele Abkürzungen, z.B. "Inhaber.Tabelle" oder "Tabelle". In MySQL schreibt man ggf. "DB.Tabelle".

Inhalt

Terme (1)

- Terme sind syntaktische Konstrukte, die zu einem Wert ausgewertet werden können (z.B. zu einer Zahl, einer Zeichenkette, oder zu einer Zeile in einer Tabelle).
- Es gibt drei Arten von Termen:
  - Konstanten, z.B. `1`, `'abc'`,
  - Variablen, z.B. `X`,
  - zusammengesetzte Terme, bestehend aus Funktionssymbolen angewandt auf Argumentterme, z.B. `NACHNAME(S)`.

    Zusammengesetzte Terme können beliebig weit verschachtelt werden, z.B. `(PUNKTE(B)/MAXPT(A)) * 100`.
- Terme sollten aus Programmiersprachen bekannt sein. Dort sagt man (Wert-)Ausdruck/Expression statt Term.

Terme (2)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Die Menge $TE_{\Sigma,\nu}(s)$ der Terme der Sorte s bezüglich (Σ, ν) ist folgendermaßen rekursiv definiert:
  - Jede Variable $V \in VARS$ mit $\nu(V) = s$ ist ein Term der Sorte s (dafür muss ν definiert sein für V).
  - Jede Konstante $c \in \mathcal{F}_{\epsilon,s}$ ist ein Term der Sorte s.
  - Wenn t_1 ein Term der Sorte s_1 ist, ..., t_n ein Term der Sorte s_n, und $f \in \mathcal{F}_{\alpha,s}$ mit $\alpha = s_1 \ldots s_n$, $n \geq 1$, dann ist $f(t_1, \ldots, t_n)$ ein Term der Sorte s.

Terme (3)

Definition, fortgesetzt:

- Jeder Term kann durch endlich häufige Anwendung obiger Regeln konstruiert werden. Nichts anderes ist ein Term.

  Diese Bemerkung ist formal wichtig, da die obigen Regeln nur festlegen, was ein Term ist, und nicht, was kein Term ist. Dazu muss die Definition abgeschlossen werden. (Selbst wenn man die obigen Regeln als Gleichungssystem auffasst, müßten unendliche Baumstrukturen als Lösungen ausgeschlossen werden.)

Definition:

- $TE_{\Sigma,\nu} := \bigcup_{s \in \mathcal{S}} TE_{\Sigma,\nu}(s)$ sei die Menge aller Terme.

Terme (4)

- Einige Funktionen werden üblicherweise als Infix-Operatoren zwischen ihre Argumente (Operanden) geschrieben, also z.B. `X+1` statt der "offiziellen" Notation `+(X, 1)`.

  Wenn man damit anfängt, muss man auch Rangfolgen (Prioritäten) der Operatoren definieren, und explizite Klammerung erlauben. Die formalen Definitionen werden dadurch komplizierter.

- Aufrufe von Funktionen der Stelligkeit 1 kann man auch in Punktnotation (objektorientiert) schreiben, z.B. "`S.NACHNAME`" für "`NACHNAME(S)`".

  "Syntaktischer Zucker" wie Infix- und Punktnotation ist in der Praxis sinnvoll, aber für die Theorie der Logik nicht wichtig. In Programmiersprachen gibt es manchmal Unterschiede zwischen der "konkreten Syntax" und der "abstrakten Syntax" (Syntaxbaum). Die abstrakte Syntax läßt viele Details weg und beschreibt eher die internen Datenstrukturen des Compilers.

Wert eines Terms

Definition:

- Seien eine Signatur Σ, eine Variablendeklaration ν für Σ, eine Σ-Interpretation $\mathcal{I}$, und eine Variablenbelegung $\mathcal{A}$ für $(\mathcal{I}, \nu)$ gegeben.
- Der Wert $\langle \mathcal{I}, \mathcal{A} \rangle [t]$ eines Terms $t \in TE_{\Sigma,\nu}$ ist wie folgt definiert (Rekursion über die Termstruktur):
  - Ist t eine Konstante c, dann ist $\langle \mathcal{I}, \mathcal{A} \rangle [t] := \mathcal{I}[c]$.
  - Ist t eine Variable V, dann ist $\langle \mathcal{I}, \mathcal{A} \rangle [t] := \mathcal{A}(V)$.
  - Hat t die Form $f(t_1, \ldots, t_n)$, mit t_i der Sorte s_i: $\langle \mathcal{I}, \mathcal{A} \rangle [t] := \mathcal{I}[f, s_1 \ldots s_n](\langle \mathcal{I}, \mathcal{A} \rangle [t_1], \ldots, \langle \mathcal{I}, \mathcal{A} \rangle [t_n])$.

    D.h. man wertet die Argumentterme aus und wendet auf die Ergebnisse die Funktion an, durch die das Funktionssymbol f interpretiert wird.

Term/Skalare Ausdrücke in SQL (1)

- Ein Term (skalarer Ausdruck) in SQL kann zu einem Wert eines Datentyps ausgewertet werden.

  Der Begriff "Term" wird in der Logik verwendet. In Programmiersprachen sagt man "Ausdruck". Der SQL-Standard verwendet "skalarer Ausdruck", weil es dort auch "Tabellenausdrücke" gibt.

- Drei wichtige Arten von Termen in SQL sind (es gibt mehr):
  - Attribut-Referenzen, z.B. `STUDENTEN.SID`.
  - Konstanten ("Literale"), z.B. `'Lisa'`, `1`.
  - Zusammengesetzte Terme, z.B. `0.9 * MAXPT`.

    Zusammengesetzte Terme bestehen aus Teil-Termen, die verknüpft werden mit Datentyp-Operatoren wie `+`, `-`, `*`, `/` (für Zahlen), `||` (String-Konkatenation) und Datentyp-Funktionen wie `SIN`.

Term/Skalare Ausdrücke in SQL (2)

- Terme verwendet man in Bedingungen, z.B. enthält

  `B.PUNKTE` > `A.MAXPT * 0.8`

  die Terme "`B.PUNKTE`" und "`A.MAXPT * 0.8`".

- Auch `SELECT`-Liste kann beliebige Terme enthalten:

```
SELECT NACHNAME || ', ' || VORNAME
FROM   STUDENTEN
```

...
Weiss, Lisa
Grau, Michael
Sommer, Daniel
Winter, Iris

Attribut-Referenzen (1)

- Auf Attribute kann man in dieser Form zugreifen:

  `Variable.Attribut`

- Hat nur eine Tupelvariable das Attribut, darf der Variablenname fehlen. Z.B. ist diese Anfrage legal:

```
SELECT ATYP, ANR, PUNKTE
FROM   STUDENTEN S, BEWERTUNGEN B
WHERE  S.SID = B.SID
AND    VORNAME = 'Lisa' AND NACHNAME = 'Weiss'
```

  "VORNAME" und "NACHNAME" gibt es nur in STUDENTEN — daher ist klar, dass die Variable "S" gemeint ist.
  "ATYP", "ANR" und "PUNKTE" können entsprechend nur "B" zugeordnet werden.
  "SID" allein wäre jedoch mehrdeutig, da sowohl "S" als auch "B" ein Attribut mit diesem Namen haben.

Attribut-Referenzen (2)

- Gegeben sei diese Anfrage:

```
SELECT ANR, SID, PUNKTE, MAXPT
FROM   BEWERTUNGEN B, AUFGABEN A
WHERE  B.ANR = A.ANR
AND    B.ATYP = 'H' AND A.ATYP = 'H'
```

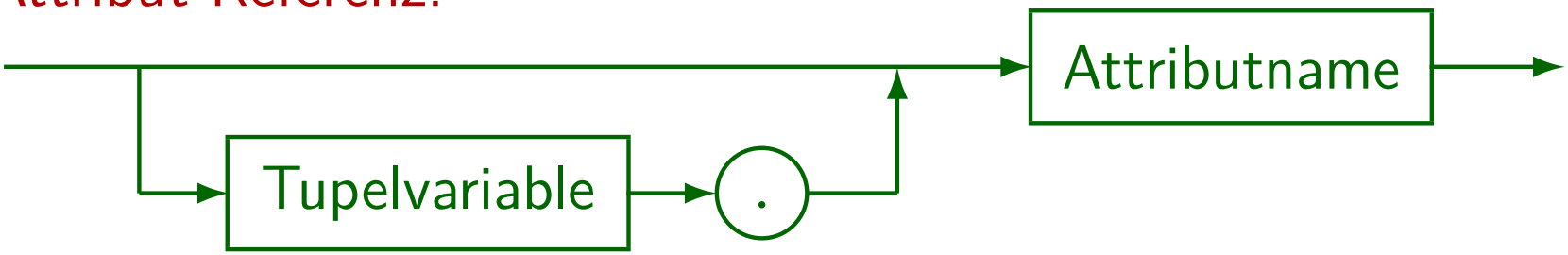
Falsch!

- SQL verlangt, dass der Nutzer festlegt, ob `A.ANR` oder `B.ANR` unter `SELECT` gemeint ist, obwohl beide gleich sind, so dass es eigentlich egal wäre.

  Die Regel ist rein syntaktisch: Hat mehr als eine Tupelvariable in der `FROM`-Klausel das Attribut "`ANR`", darf die Tupelvariable nicht fehlen oder das DBMS (z.B. Oracle) wird den Fehler "ORA-00918: column ambiguously defined" ausgeben. DB2, SQL Server, Access, MySQL sind auch so streng.

Attribut-Referenzen (3)

Attribut-Referenz:



- Wie oben erläutert, darf die Tupelvariable nur weggelassen werden, wenn es in der Variablendeklaration, in deren Gültigkeitsbereich die Attribut-Referenz steht, nur eine Tupelvariable über einer Tabelle gibt, die ein Attribut mit dem Namen hat. D.h. man muss die Tupelvariable eindeutig rekonstruieren können. Diese Regel wird später im Zusammenhang mit Unteranfragen noch etwas verfeinert, s. Kap. 7.

Tupelvariable:

Attributname:

Zusammengesetzte Terme (1)

- Der SQL-86-Standard enthielt nur `+`, `-`, `*`, `/`.
- Derzeitige DBMS unterscheiden sich immer noch in anderen Datentyp-Operationen (siehe Kapitel 4).

  Aber sie haben meist eine große Auswahl an Datentyp-Operationen, z.B. `sin`, `cos`, `substr`. Kapitel 4 enthält Listen von Datentyp-Operationen für verschiedene Systeme.

- Z.B. ist der Operator `||` im SQL-92-Standard enthalten, aber funktioniert z.B. nicht in SQL Server.

  String-Konkatenation wird in SQL Server und Access "+" geschrieben.
  In MySQL muss man "`concat(`s_1`,` s_2`)`" schreiben (aber es gibt `--ansi`).
  Andere Datentyp-Funktionen (z.B. `SUBSTR`) sind sogar noch weniger standardisiert.

Zusammengesetzte Terme (2)

- SQL kennt die Standard-Vorrangregeln,
  z.B. bedeutet `A+B*C`:

  `A+(B*C)`,

  und nicht

  `(A+B)*C`.

- Klammern `(...)` können verwendet werden, um eine bestimmte Struktur zu erzwingen.

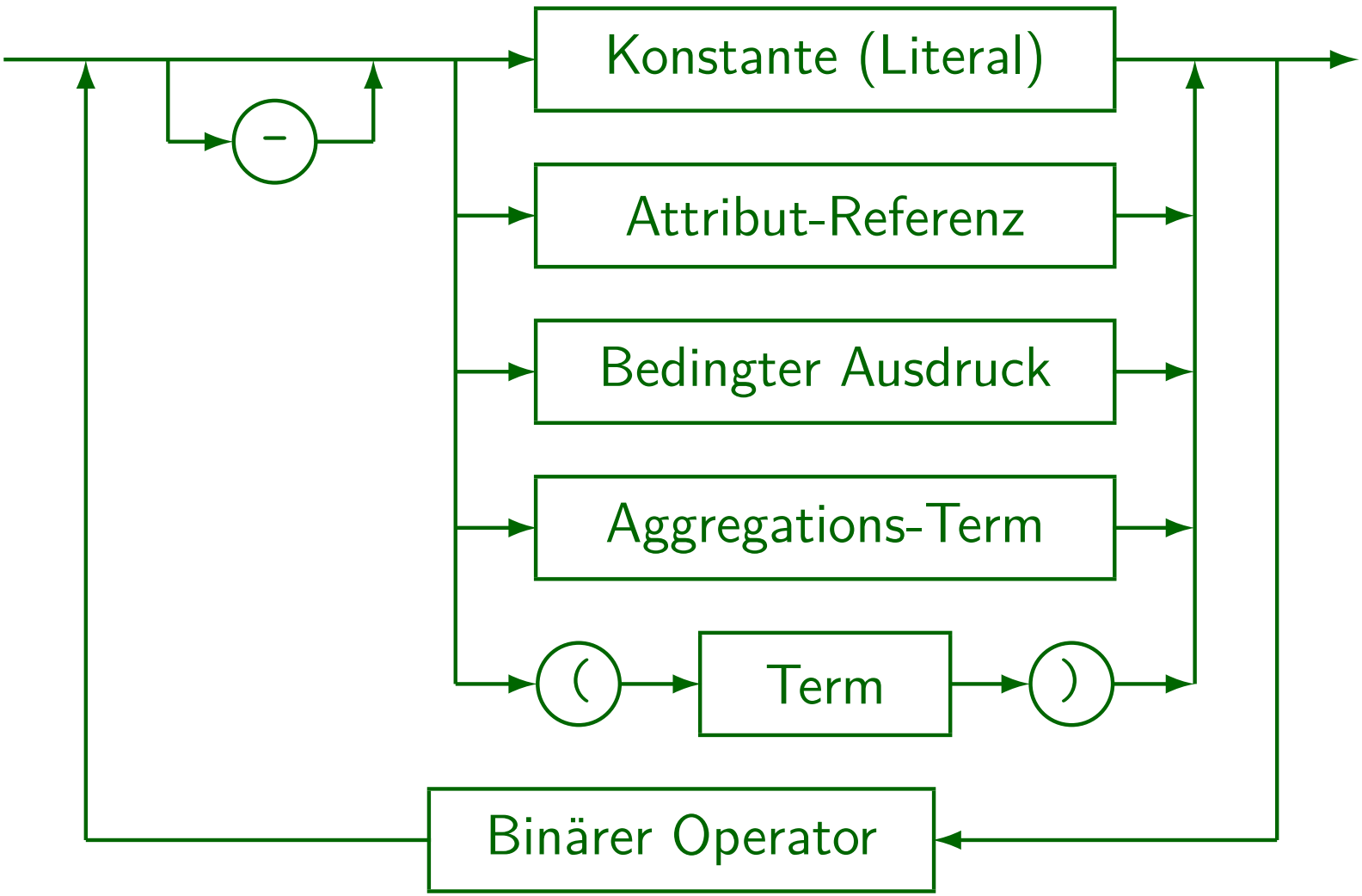
- Übung: Was ist das Ergebnis von `7+3*2-4-1`?

  Es kann nützlich sein, einen Operator-Baum zu zeichnen.

  "`-`" ist links-assoziativ (von links ausgewertet).

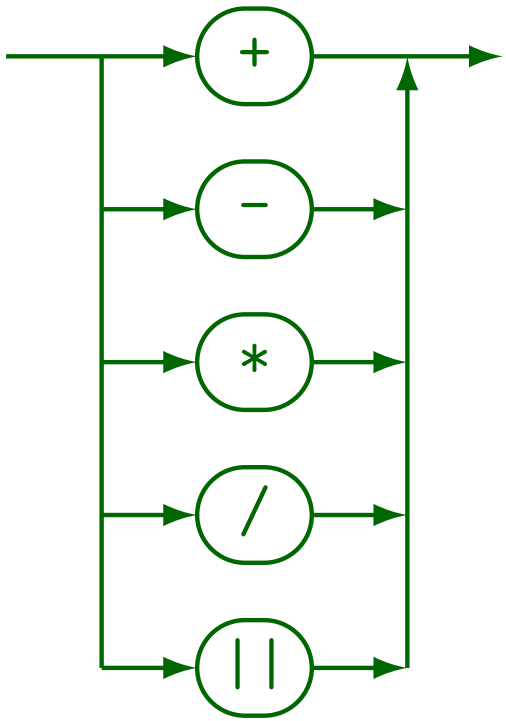
Terme: Syntax (1)

Term (Skalarer Ausdruck, Wert-Ausdruck):



Terme: Syntax (2)

Binärer Operator:



- SQL Server, Access, MySQL verwenden nicht "||" für die Konkatenation.
- Punktrechnung (*, /) bindet stärker als Strichrechnung (+, -).
- Die implizite Klammerung ist von links, z.B. wird `A-B-C` als `(A-B)-C` verstanden.

SELECT-Klausel, *

- SELECT legt die Terme fest, die ausgegeben werden, falls die WHERE-Bedingung wahr ist (Ergebnis-Spalten).
- SELECT * kann verwendet werden, um alle Spalten der Tabelle(n) unter FROM auszugeben, z.B. ist

```
SELECT *
FROM   STUDENTEN
```

  äquivalent zu

```
SELECT SID, VORNAME, NACHNAME, EMAIL
FROM   STUDENTEN
```

- In Programmen sollte man * vermeiden, da später manchmal Spalten zu Tabellen hinzugefügt werden.

Umbenennung von Spalten

- Beispiel für Umbenennung von Ausgabe-Spalten:

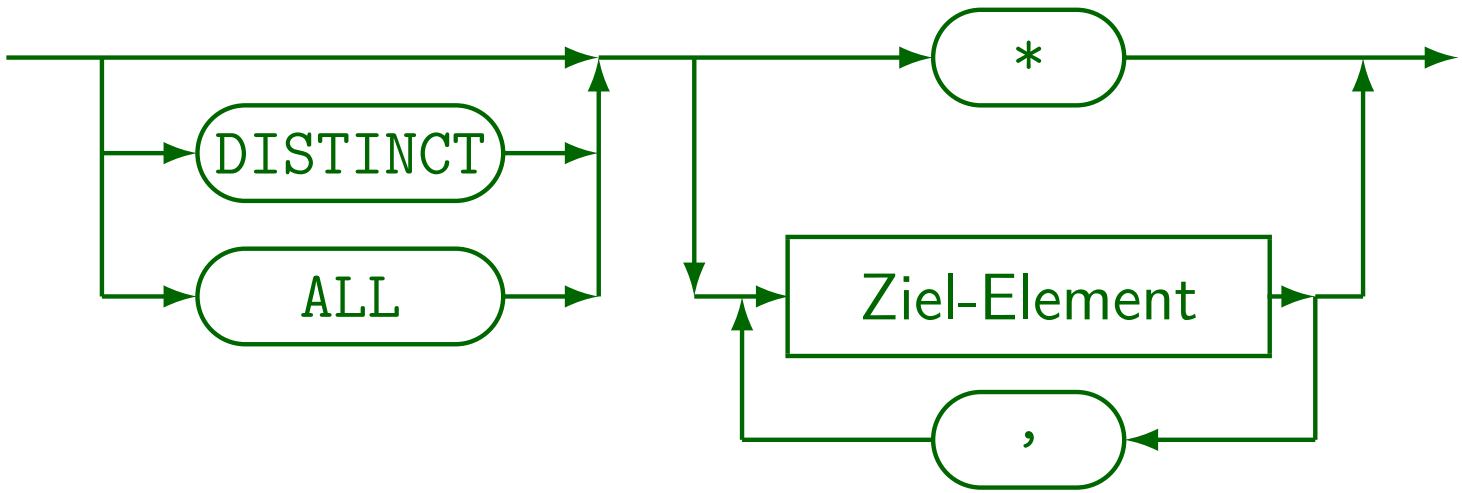
```
SELECT VORNAME AS V_Name, NACHNAME AS "Name"
FROM   STUDENTEN
```

V_NAME	Name
Lisa	Weiss
Michael	Grau
Daniel	Sommer
Iris	Winter

- Dies funktioniert in SQL-92, Oracle, PostgreSQL, DB2, SQL Server, MySQL/MariaDB, Access, aber nicht in SQL-86.
- "AS" kann in SQL-92 und allen obigen Systemen außer Access weggelassen werden (es ist ein "noise word").

SELECT-Syntax (1)

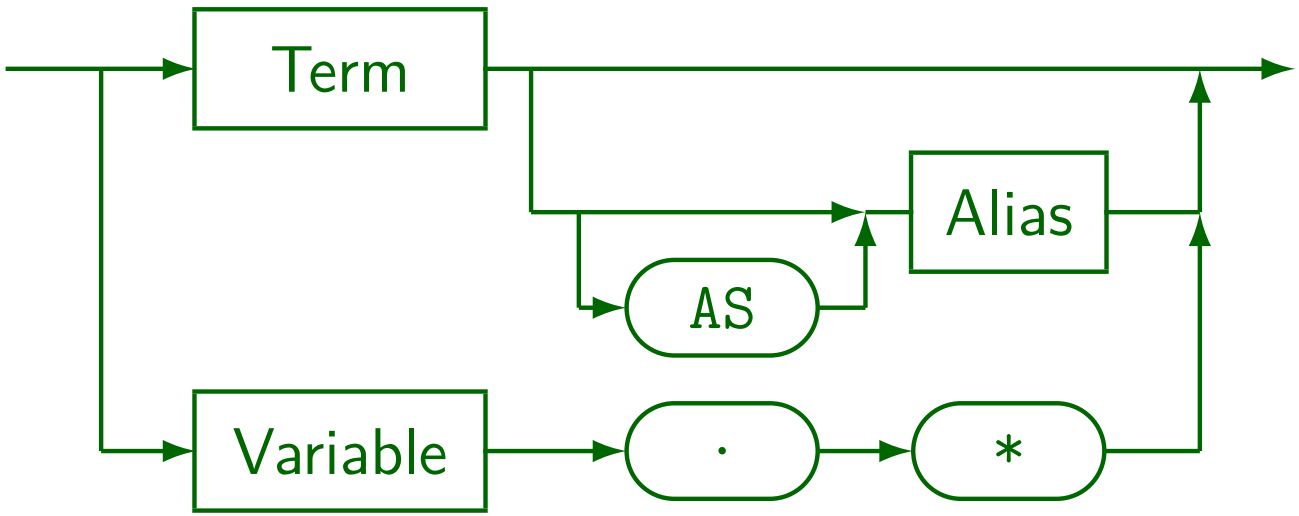
Ziel-Liste (nach SELECT):



- Duplikateliminierung (mit DISTINCT) wird in Kapitel 6 besprochen.
- ALL (keine Duplikat-Elimination) ist der Default.
  Es erfolgt eine Ausgabe der Terme unter SELECT für jede Variablenbelegung, bei der die WHERE-Bedingung wahr ist. Dabei ist möglich, dass mehrfach die gleiche Ergebniszeile ausgegeben wird.

SELECT-Syntax (2)

Ziel-Element:



- "Variable.*" und "[AS] Alias" funktionieren in SQL-92, Oracle, SQL Server, DB2, MySQL und Access (in Access wird "AS" benötigt). Diese Konstruktionen sind im alten SQL-86-Standard nicht enthalten.

Inhalt

Atomare Formeln (1)

- Formeln sind syntaktische Ausdrücke, die zu einem Wahrheitswert ausgewertet werden können, z.B

$$1 \leq X \ \wedge \ X \leq 10.$$

- Atomare Formeln sind die grundlegenden Bestandteile zur Bildung dieser Formeln (Vergleiche etc.).
- Atomare Formeln können folgende Formen haben:
  - Prädikatsymbol, angewandt auf Terme, z.B. `B.PUNKTE > 5` oder `S.NACHNAME = 'Weiss'`.

    Man kann "=" auch fest in die Logik einbauen. Da in SQL aber die Gleichheit von Zeichenketten recht kompliziert ist, muss man es hier als Prädikat verstehen, das vom DBMS interpretiert wird.
  - Logischen Konstante: $\top$ (wahr) und $\bot$ (falsch).

    Diese gibt es nicht in SQL. Man kann z.B. `1=1` für "wahr" schreiben.

Atomare Formeln (2)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Eine atomare Formel ist ein Ausdruck der Form:
  - $p(t_1, \ldots, t_n)$ mit $p \in \mathcal{P}_\alpha$, $\alpha = s_1 \ldots s_n \in \mathcal{S}^*$, $n > 0$ und $t_i \in TE_{\Sigma,\nu}(s_i)$ für $i = 1, \ldots, n$.

    Für einige Prädikate (z.B. <) ist die Infixnotation üblich.
  - p mit $p \in \mathcal{P}_\epsilon$,
  - $\top$ oder $\bot$.
- $AT_{\Sigma,\nu}$ sei die Menge der atomaren Formeln für Σ, ν.

Formeln (1)

Definition:

- Sei eine Signatur $\Sigma = (\mathcal{S}, \mathcal{P}, \mathcal{F})$ und eine Variablendeklaration ν für Σ gegeben.
- Die Menge $QFF_{\Sigma,\nu}$ der quantorenfreien (Σ, ν)-Formeln ist folgendermaßen rekursiv definiert:
  - Jede atomare Formel $F \in AT_{\Sigma,\nu}$ ist eine Formel.
  - Wenn F und G Formeln sind, so auch $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \leftarrow G)$, $(F \rightarrow G)$, $(F \leftrightarrow G)$.

    Nach dieser Definition müssen viele Klammern gesetzt werden, um eine eindeutige syntaktische Struktur zu sichern. Das ist formal einfach, aber für die praktische Anwendung ist diese Syntax unpraktisch. Auf der nächsten Folie finden sich Regeln zur Einsparung von Klammern.
  - Nichts sonst ist eine quantorenfreie Formel.

Formeln (2)

- Die intuitive Bedeutung der Formeln ist wie folgt:
  - $\neg F$: "nicht F" (F ist falsch).
  - $F \wedge G$: "F und G" (beide sind wahr).
  - $F \vee G$: "F oder G" (eine oder beide sind wahr).
  - $F \leftarrow G$: "F wenn G" (ist G wahr, so auch F)
  - $F \rightarrow G$: "wenn F, dann G"
  - $F \leftrightarrow G$: "F genau dann, wenn G".
- Regeln zur Einsparung von Klammern:
  - Die äußersten Klammern sind nie notwendig.
  - $\neg$ bindet am stärksten, dann $\wedge$, dann $\vee$, dann $\leftarrow$, $\rightarrow$, $\leftrightarrow$.
  - Bei gleicher Bindungsstärke wird von links geklammert.

SQL-Anfragen aus Sicht der Logik

- Der Aufbau einer einfachen SQL-Anfrage ist:

  $$\begin{array}{ll} \texttt{SELECT} & t_1, \ \ldots, \ t_k \\ \texttt{FROM} & s_1 \ X_1, \ \ldots, \ s_n \ X_n \\ \texttt{WHERE} & F \end{array}$$

- Sei ν die Variablendeklaration aus der `FROM`-Klausel: $\nu = \{X_1/s_1, \ldots, X_n/s_n\}$.
- Sei Σ die Signatur, die sich aus dem Datenbank-Schema und den in das DBMS eingebauten Datentypen ergibt.
- Dann sind $t_1, \ldots, t_k$ Terme bezüglich (Σ, ν).

  Die Ergebnissorte des Terms muss eine Datensorte sein (keine Tupelsorte).

- F ist eine Formel bezüglich (Σ, ν).

  SQL versteht zum Spaltenzugriff nur die Notation `X.A`, nicht `A(X)`.

DB-Anfragen aus Sicht der Logik (1)

- Der einfachste Ansatz ist:
  - Eine Anfrage ist eine Formel F.

    Die Signatur ist wie oben durch die Datenbank gegeben.
    Die Variablendeklaration muss man aber eventuell noch dazu schreiben.
    Ohne Überladen würde sich die Sorte jeder Variable aus der Verwendung ergeben, dann müssten Variablen nicht explizit deklariert werden.
  - Gesucht sind nun Variablenbelegungen $\mathcal{A}$, die die Formel F in der Interpretation $\mathcal{I}$ wahr machen, die durch die DBMS-Implementierung und den DB-Zustand gegeben ist.

    Die Menge aller solcher Belegungen ist die Antwort auf die Anfrage.
    Man muss die Formeln so einschränken, dass diese Menge immer endlich ist ("bereichsbeschränke Formeln"). Bei SQL ist das kein Problem, weil die Variablen nur über endlich vielen Tabellenzeilen laufen.

DB-Anfragen aus Sicht der Logik (2)

- "Anfragen sind Formeln" funktioniert so im Bereichskalkül.

  Es funktioniert auch beim theoretischen Tupelkalkül: Dort gibt es nur eine Antwort-Variable, deren Belegungen mit Tupeln ausgedruckt werden.

  Man braucht dann aber sehr schnell Quantoren für andere Tupelvariablen.

- Beim SQL-Tupelkalkül ist das Problem, dass man aus den Belegungen für oft mehrere Variablen $X_1, \ldots, X_n$, die über Tabellenzeilen laufen, eine Antwortzeile konstruieren muss.

  Man will meist auch gar nicht die ganzen Tupel ausdrucken.

- Das ist der Zweck der Terme $t_1, \ldots, t_k$, die eine Belegung, die die Formel F wahr macht, in ein Antwort-Tupel umrechnen:

$$\{t_1, \ldots, t_k \; [s_1 \, X_1, \ldots, s_n \, X_n] \mid F\}.$$

  Dies entspricht genau

  `SELECT` t_1`,` $\ldots$`,` t_k `FROM` s_1 X_1`,` $\ldots$`,` s_n X_n `WHERE` F.

  $t_1, \ldots, t_k$ sind Terme und F eine Formel bzgl. (Σ, ν) mit $\nu := \{X_1/s_1, \ldots, X_n/s_n\}$.

Bedingungen in SQL (1)

- Es gibt in SQL nur drei logische Junktoren:
  - `AND` wird für $\wedge$ geschrieben (Konjunktion).
  - `OR` wird für $\vee$ geschrieben (Disjunktion).
  - `NOT` wird für $\neg$ geschrieben (Negation).
- Bedingungen bestehen aus atomaren Formeln, z.B.

  ```
  PUNKTE >= 8,
  ```

  verbunden mit "`AND`", "`OR`", "`NOT`".
- `AND` bindet stärker als `OR`, somit wird

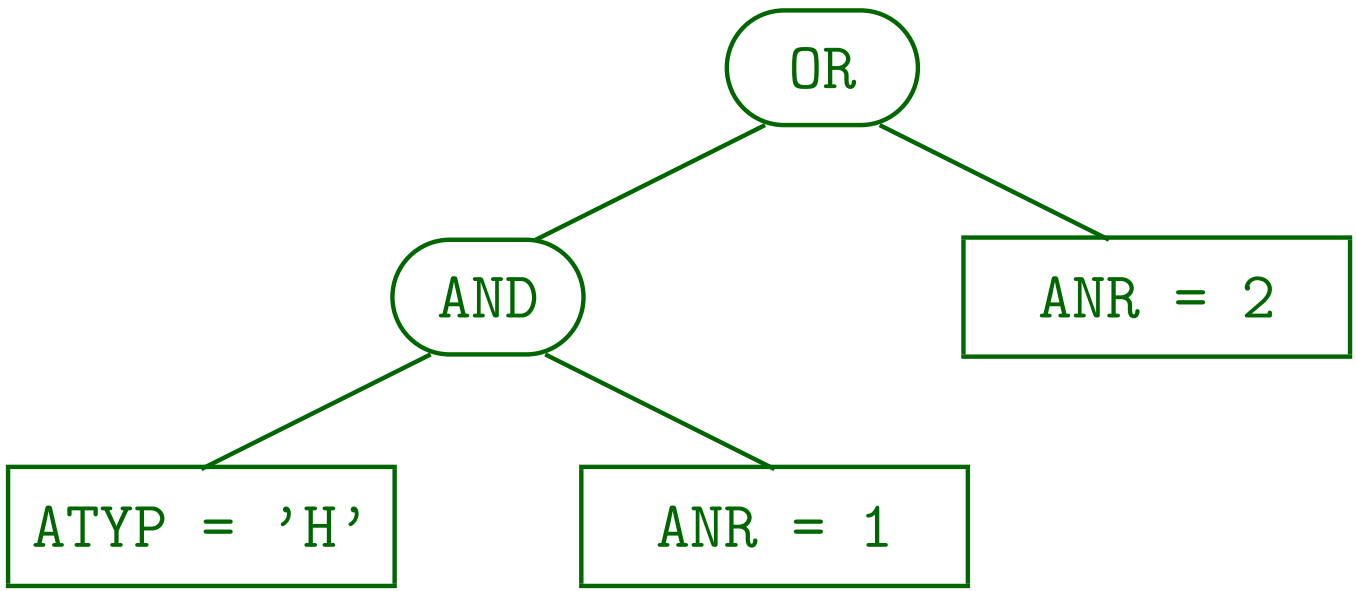
  ```
  ATYP = 'H' AND ANR = 1 OR ANR = 2
  ```

  implizit so geklammert:

  ```
  (ATYP = 'H' AND ANR = 1) OR ANR = 2
  ```

Bedingungen (2)

- In dem obigen Beispiel ist die implizite Klammerung vermutlich nicht die gewünschte Struktur der Anfrage.
- Es kann helfen, komplexe Bedingungen oder Terme als "Operator-Baum" darzustellen:



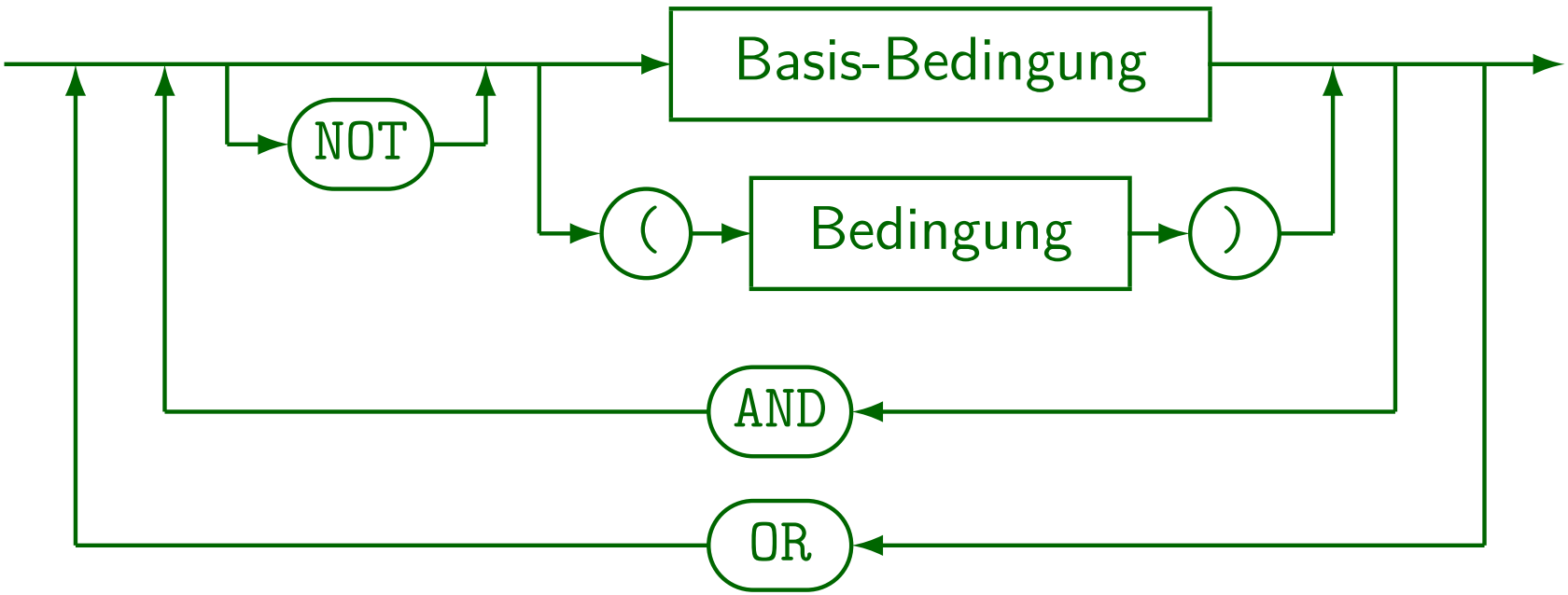
Bedingungen (3)

- `NOT` bindet am stärksten, d.h. es gilt nur für die direkt folgende Bedingung (atomare Formel).
- Klammern `( ...)` können verwendet werden, um die Bindungsstärken / Prioritäten der Operatoren aufzuheben.
- Manchmal ist es klarer, Klammern zu verwenden, auch wenn sie nicht nötig wären, um die richtige Struktur der Bedingung zu erhalten.

  Anfänger neigen jedoch dazu, viele Klammern zu verwenden (wahrscheinlich weil sie sich über die Bindungsstärken nicht sicher sind). Das macht die Formel nicht verständlicher.

Bedingungen (4)

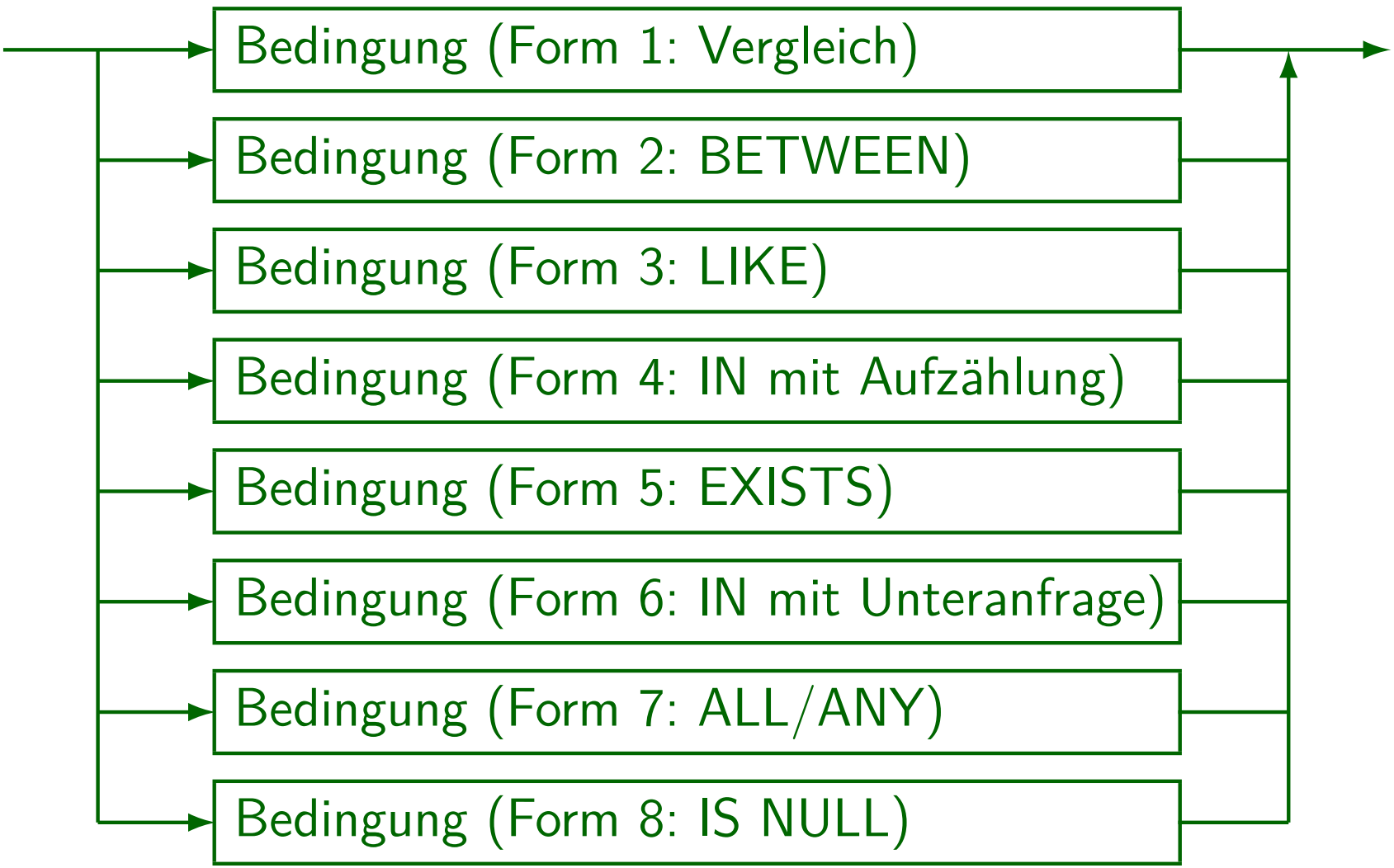
Bedingung:



- SQL-92 erlaubt "`IS NOT TRUE`", "`IS FALSE`" usw. nach Formeln (nicht in Oracle 8.0, SQL Server, DB2, MySQL, Access unterstützt).
- Die syntaktische Kategorie "Basis-Bedingung" enthält natürlich insbesondere die atomaren Formeln, aber auch Bedingungen mit Unteranfragen (entsprechen quantifizierten Formeln, siehe Kapitel 7).

Bedingungen (5)

Basis-Bedingung:



Bedingungen (6)

- `AND` und `OR` müssen auf beiden Seiten vollständige logische Bedingungen haben (etwas, das wahr oder falsch ist).
- Somit ist Folgendes ein Syntaxfehler, obwohl es in der natürlichen Sprache erlaubt wäre:

```
SELECT DISTINCT SID
FROM   BEWERTUNGEN
WHERE  ATYP = 'H' AND PUNKTE >= 9
AND    ANR = 1 OR 2
```

Falsch!

- Ausnahme: `... BETWEEN ... AND ...`

  Hier bezeichnet das Wort `AND` keinen logischen Operator.

  Diese Form der Bedingung wird erst in Kapitel 6 eingeführt.

Vergleiche (1)

Bedingung (Form 1: Vergleich):

- Vergleichsoperatoren: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Man kann sie sowohl für Zahlen als auch für Strings verwenden, z.B.: `PUNKTE >= 8`, `NACHNAME < 'M'`.
- "Ungleich" wird in SQL als "`<>`" geschrieben.

  Oracle, SQL Server, DB2 und MySQL verstehen auch "`!=`" (Access nicht). "`^=`" funktioniert in Oracle und DB2, aber nicht in SQL Server, Access oder MySQL.

Vergleiche (2)

- Zahlen werden anders verglichen als Zeichenketten, z.B. `3 < 20`, aber `'3' > '20'`.

  Strings werden Zeichen für Zeichen verglichen, bis das Ergebnis klar ist.
  In diesem Fall kommt "3" alphabetisch nach "2", der Rest der Strings ist egal.

- Nach dem SQL-92-Standard ist es falsch, Zeichenketten mit Zahlen zu vergleichen, z.B. `3 > '20'`.

  In DB2 und Access gibt es tatsächlich den Typfehler. PostgreSQL wandelt eine String-Konstante bei Bedarf in eine Zahl um (sofern das Format numerisch ist), aber nicht eine String-wertige Spalte. Oracle und MySQL konvertieren den String in eine Zahl und vergleichen numerisch. Bei Oracle gibt es dann einen Laufzeitfehler, wenn das Format nicht numerisch ist. MySQL konvertiert den String in diesem Fall in 0, z.B. ist `0 = 'abc'` in MySQL wahr. Bei MS SQL Server setzt sich beim Vergleich von Spalte und Konstante der Spaltentyp durch, ansonsten der numerische Typ.

Inhalt

Wahrheit einer quantorenfreien Formel (1)

Definition:

- Der Wahrheitswert $\langle \mathcal{I}, \mathcal{A} \rangle [F] \in \{\mathbf{f}, \mathbf{w}\}$ einer Formel F in $\langle \mathcal{I}, \mathcal{A} \rangle$ ist definiert als (**f** bedeutet falsch, **w** wahr):
  - Ist F eine atomare Formel $p(t_1, \ldots, t_n)$ mit den Termen t_i der Sorte s_i:

$$\langle \mathcal{I}, \mathcal{A} \rangle [F] := \begin{cases} \mathbf{w} & \text{falls } (\langle \mathcal{I}, \mathcal{A} \rangle [t_1], \ldots, \langle \mathcal{I}, \mathcal{A} \rangle [t_n]) \\ & \qquad \in \mathcal{I}[p,\ s_1 \ldots s_n] \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - Ist F "true" $\top$: $\langle \mathcal{I}, \mathcal{A} \rangle [F] := \mathbf{w}$.
  - Ist F "false" $\bot$: $\langle \mathcal{I}, \mathcal{A} \rangle [F] := \mathbf{f}$.
  - (auf der nächsten Folie fortgesetzt ...)

Wahrheit einer quantorenfreien Formel (2)

- Wahrheitswert einer Formel, fortgesetzt:
  - Hat F die Form $(\neg G)$:

$$\langle \mathcal{I}, \mathcal{A} \rangle[F] := \begin{cases} \mathbf{w} & \text{falls } \langle \mathcal{I}, \mathcal{A} \rangle[G] = \mathbf{f} \\ \mathbf{f} & \text{sonst.} \end{cases}$$

  - Hat F die Form $(G_1 \wedge G_2)$, $(G_1 \vee G_2)$, etc.:

G_1	G_2	$\wedge$	$\vee$	$\leftarrow$	$\rightarrow$	$\leftrightarrow$
f	**f**	**f**	**f**	**w**	**w**	**w**
f	**w**	**f**	**w**	**f**	**w**	**f**
w	**f**	**f**	**w**	**w**	**f**	**f**
w	**w**	**w**	**w**	**w**	**w**	**w**

Beispiel zum Lesen der Tabelle (dritte Zeile, Spalte "$\wedge$"):
Falls $\langle \mathcal{I}, \mathcal{A} \rangle[G_1] = \mathbf{w}$ und $\langle \mathcal{I}, \mathcal{A} \rangle[G_2] = \mathbf{f}$, dann $\langle \mathcal{I}, \mathcal{A} \rangle[(G_1 \wedge G_2)] = \mathbf{f}$.

Modell (1)

Definition:

- Ist $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathbf{w}$, so schreibt man auch $\langle \mathcal{I}, \mathcal{A} \rangle \models F$. Dann heißt $\langle \mathcal{I}, \mathcal{A} \rangle$ ein Modell von F.

  Man sagt dann auch $\langle \mathcal{I}, \mathcal{A} \rangle$ erfüllt F bzw. F ist in $\langle \mathcal{I}, \mathcal{A} \rangle$ wahr.

  Viele Autoren wenden den Beriff "Modell" allerdings nur auf Interpretationen allein an (ohne Variablenbelegung), wie im folgenden Punkt beschrieben.

- Sei F eine (Σ, ν)-Formel. Gilt $\langle \mathcal{I}, \mathcal{A} \rangle[F] = \mathbf{w}$ für alle Variablenbelegungen $\mathcal{A}$ (für $\mathcal{I}$ und ν), so schreibt man $\mathcal{I} \models F$ und nennt $\mathcal{I}$ ein Modell von F.

  D.h. freie Variablen werden als $\forall$-quantifiziert behandelt.

  Die Variablenbelegung ist aber irrelevant, falls F eine geschlossene Formel ist.

- $\langle \mathcal{I}, \mathcal{A} \rangle$ heißt Modell einer Formelmenge Φ gdw. $\langle \mathcal{I}, \mathcal{A} \rangle \models F$ für alle Formeln $F \in \Phi$.

Modell (2)

Definition:

- Eine Formel F heißt konsistent gdw. es $\mathcal{I}$ und $\mathcal{A}$ gibt mit $\langle \mathcal{I}, \mathcal{A} \rangle \models F$ (d.h. wenn sie ein Modell hat).

  Entsprechend für Mengen von Formeln.

  Manche Autoren nennen eine Formel nur dann konsistent, wenn sie in einer Interpretation $\mathcal{I}$ für alle Variablenbelegungen $\mathcal{A}$ wahr ist. Wenn sie nur für mindestens eine Variablenbelegung wahr ist, würde die Formel "erfüllbar" heißen.

- F ist inkonsistent gdw. F ist nicht konsistent.

  D.h. F ist immer falsch, egal welche Interpretation und Variablenbelegung man nimmt. Mit anderen Worten: F hat kein Modell.
  (Man beachte wieder, dass es in manchen Büchern "unerfüllbar" heißt.)

- F heißt Tautologie gdw. $\langle \mathcal{I}, \mathcal{A} \rangle \models F$ für alle $\mathcal{I}$ und $\mathcal{A}$.

  Eine Taulogie ist also immer wahr.

Bedingungen (3)

- "Geben Sie die SIDs von Lisa und Iris aus".

```
SELECT SID
FROM   STUDENTEN
WHERE  VORNAME = 'Lisa' AND VORNAME = 'Iris'
```

Falsch!

STUDENTEN		
SID	VORNAME	NACHNAME
101	Lisa	Weiss
102	Michael	Grau
103	Daniel	Sommer
104	Iris	Winter

VORN.='Lisa'	VORN.='Iris'	WHERE
wahr	falsch	falsch
falsch	falsch	falsch
falsch	falsch	falsch
falsch	wahr	falsch

- Die obige Bedingung ist inkonsistent.
  Hier muss "OR" verwendet werden, nicht "AND".

Antwort auf eine Anfrage

Definition:

- Sei die folgende Anfrage gegeben:

$$\{t_1, \ldots, t_k\ [s_1\ X_1, \ldots, s_n\ X_n] \mid F\}.$$

Oder in SQL: `SELECT` t_1 `,` $\ldots$ `,` t_k `FROM` $s_1\ X_1$ `,` $s_n\ X_n$ `WHERE` F.

$t_1, \ldots, t_k$ sind Terme und F eine Formel bzgl. (Σ, ν) mit $\nu := \{X_1/s_1, \ldots, X_n/s_n\}$.

- Sei $\mathcal{I}$ die Interpretation, die dem Datenbank-Zustand entspricht.

Der Anteil für die Symbole der Datensignatur $\Sigma_{\mathcal{D}}$ ist natürlich die in das DBMS eingebaute Interpretation $\mathcal{I}_{\mathcal{D}}$.

- Dann ist die Antwort die Menge

$$\{(\langle \mathcal{I}, \mathcal{A} \rangle[t_1], \ldots, \langle \mathcal{I}, \mathcal{A} \rangle[t_k]) \mid \mathcal{A} \text{ ist Variablenbelegung für } \mathcal{I} \text{ und } \nu \text{ mit } \langle \mathcal{I}, \mathcal{A} \rangle \models F\}.$$

Inhalt

Implikation

Definition:

- Eine Formel oder Menge von Formeln Φ impliziert (logisch) eine Formel oder Menge von Formeln G gdw. jedes Modell $\langle \mathcal{I}, \mathcal{A} \rangle$ von Φ auch ein Modell von G ist. In diesem Fall schreibt man $\Phi \models G$.

  In der Behandlung von freien Variablen unterscheiden sich die Definitionen verschiedener Autoren. Z.B. würde $X = Y \wedge Y = Z \models X = Z$ nach obiger Definition gelten. Betrachtet man freie Variablen aber als implizit allquantifiziert, so gilt das nicht.

  Beachte: Das Zeichen $\models$ ist überladen. Steht links eine Interpretation bzw. ein Paar aus Interpretation und Variablenbelegung, bedeutet es "ist Modell von" (d.h. die Interpretation macht die Formel rechts wahr). Steht links eine Formelmenge, bedeutet es die logische Implikation: Jedes Modell der Formelmenge links macht die Formel rechts wahr.

Äquivalenz (1)

Definition:

- Zwei (Σ, ν)-Formeln oder Mengen solcher Formeln F_1 und F_2 heißen (logisch) äquivalent gdw. sie die gleichen Modelle haben, d.h. wenn für jede Σ-Interpretation $\mathcal{I}$ und jede $(\mathcal{I}, \nu)$-Variablenbelegung $\mathcal{A}$ gilt:

$$\langle \mathcal{I}, \mathcal{A} \rangle \models F_1 \iff \langle \mathcal{I}, \mathcal{A} \rangle \models F_2.$$

  Man schreibt dann: $F_1 \equiv F_2$.

  Wie schon beim Modellbegriff und der logischen Implikation behandeln manche Autoren freie Variablen als implizit allquantifiziert.

  Bei Datenbanken bezieht sich "Äquivalenz" häufig auf eine gegebene Menge von Integritätsbedingungen: Dann werden nicht beliebige Σ-Interpretationen $\mathcal{I}$ betrachtet, sondern nur solche, die die Integritätsbedingungen erfüllen.

Äquivalenz (2)

Lemma:

- F_1 und F_2 sind äquivalent gdw. $F_1 \models F_2$ und $F_2 \models F_1$.
- "Äquivalenz" von Formeln ist eine Äquivalenzrelation, d.h. sie ist reflexiv, symmetrisch und transitiv.

  Reflexiv: $F \equiv F$.

  Symmetrisch: Wenn $F \equiv G$, dann $G \equiv F$.

  Transitiv: Wenn $F_1 \equiv F_2$ und $F_2 \equiv F_3$, dann $F_1 \equiv F_3$.

- Entsteht G_1 aus G_2 durch Ersetzen der Teilformel F_1 durch F_2, und gilt $F_1 \equiv F_2$, so gilt auch $G_1 \equiv G_2$.
- Wenn $F \models G$, dann $F \wedge G \equiv F$.

Einige Äquivalenzen (1)

- Kommutativität (für und, oder, gdw):
  - $F \wedge G \equiv G \wedge F$
  - $F \vee G \equiv G \vee F$
  - $F \leftrightarrow G \equiv G \leftrightarrow F$
- Assoziativität (für und, oder, gdw):
  - $F_1 \wedge (F_2 \wedge F_3) \equiv (F_1 \wedge F_2) \wedge F_3$
  - $F_1 \vee (F_2 \vee F_3) \equiv (F_1 \vee F_2) \vee F_3$
  - $F_1 \leftrightarrow (F_2 \leftrightarrow F_3) \equiv (F_1 \leftrightarrow F_2) \leftrightarrow F_3$
- Z.B. muss man sich in SQL über die Reihenfolge mehrerer mit `AND` verknüpfter Bedingungen keine Gedanken machen.

Einige Äquivalenzen (2)

- Distributivgesetz:
  - $F \wedge (G_1 \vee G_2) \equiv (F \wedge G_1) \vee (F \wedge G_2)$
  - $F \vee (G_1 \wedge G_2) \equiv (F \vee G_1) \wedge (F \vee G_2)$
- Doppelte Negation:
  - $\neg(\neg F) \equiv F$
- De Morgan'sche Regeln:
  - $\neg(F \wedge G) \equiv (\neg F) \vee (\neg G)$.
  - $\neg(F \vee G) \equiv (\neg F) \wedge (\neg G)$.

Einige Äquivalenzen (3)

- Ersetzung des Implikationsoperators:
  - $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (F \leftarrow G)$
  - $F \leftarrow G \equiv G \rightarrow F$
  - $F \rightarrow G \equiv \neg F \vee G$
  - $F \leftarrow G \equiv F \vee \neg G$
- Zusammen mit den De Morgan'schen Regeln bedeutet dies, dass z.B. $\{\neg, \vee\}$ ausreichend sind, weil die anderen logischen Junktoren $\{\wedge, \leftarrow, \rightarrow, \leftrightarrow\}$ durch diese ausgedrückt werden können.
- Die Einschränkung, dass es in SQL nur `AND`, `OR`, `NOT` gibt, ist also nicht wesentlich.

Einige Äquivalenzen (4)

- Entfernung der Negation:
  - $\neg(t_1 < t_2) \equiv t_1 \geq t_2$
  - $\neg(t_1 \leq t_2) \equiv t_1 > t_2$
  - $\neg(t_1 = t_2) \equiv t_1 \neq t_2$
  - $\neg(t_1 \neq t_2) \equiv t_1 = t_2$
  - $\neg(t_1 \geq t_2) \equiv t_1 < t_2$
  - $\neg(t_1 > t_2) \equiv t_1 \leq t_2$

  Zusammen mit dem De'Morganschen Gesetz kann man die Negation bis zu den atomaren Formeln herunterschieben, und dann durch Umdrehen der Vergleichsoperatoren eliminieren.
  Das geht später auch mit Quantoren, aber man braucht dann $\exists$ und $\forall$.
  Da es in SQL nur $\exists$ gibt, kann man dort $\neg$ vor $\exists$ nicht entfernen.

Einige Äquivalenzen (5)

- Prinzip des ausgeschlossenen Dritten:
  - $F \vee \neg F \equiv \top$ (immer wahr)
  - $F \wedge \neg F \equiv \bot$ (immer falsch)
- Vereinfachung von Formeln mit den logischen Konstanten $\top$ (wahr) und $\bot$ (falsch):
  - $F \wedge \top \equiv F$ $\quad$ $F \wedge \bot \equiv \bot$
  - $F \vee \top \equiv \top$ $\quad$ $F \vee \bot \equiv F$
  - $\neg\top \equiv \bot$ $\quad$ $\neg\bot \equiv \top$

Einige Äquivalenzen (6)

- Gleichheit ist Äquivalenzrelation:
  - $t = t \equiv \top$ (Reflexivität)
  - $t_1 = t_2 \equiv t_2 = t_1$ (Symmetrie)
  - $t_1 = t_2 \wedge t_2 = t_3 \equiv t_1 = t_2 \wedge t_2 = t_3 \wedge t_1 = t_3$ (Transitivität)
- Verträglichkeit mit Funktions-/Prädikatsymbolen:
  - $f(t_1, \ldots, t_n) = t \wedge t_i = t_i' \equiv$
    $f(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n) = t \wedge t_i = t_i'$
  - $p(t_1, \ldots, t_n) \wedge t_i = t_i' \equiv$
    $p(t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_n) \wedge t_i = t_i'$

  In SQL nicht erfüllt!

  Z.B. gilt normalerweise `'a' = 'a '`, aber `'a' || '.' = 'a ' || '.'` ist falsch!