Grundlagen und Logik

Semester Nr. 1 Grundlagen und Logik bei Karl Stroetmann

Inhaltsverzeichnis

Naive Set Theorie

Definition von Sets nach Georg Cantor (1845 – 1918)

Eine Menge ist eine wohldefinierte Ansammlung von Objekten unserer Wahrnehmung oder unseres Denkens.

Komprehensions – Axiom

Ist p(x) eine Eigenschaft, die ein Objekt x halten kann, so ist die Menge M aller Objekte mit der Eigenschaft p definiert als:

$$M := \{x \mid p(x)\}$$

Russel – Antinomie

Um Paradoxa zu vermeiden, muss bei der Definition von Mengen mehr spezifiziert werden, da sonst folgendes Paradoxon auftreten kann:

$$p(x) := \neg(x \in x)$$

$$\mathbb{R} := \{x \mid \neg(x \in x)$$

Somit stellt sich die Frage $R \in R$?

$$\Leftrightarrow R \in \{x \mid \neg(x \in x)\}$$

$\Leftrightarrow \neg R \in R$! (Wiederspruch)

Auflistung der Menge

Bsp.: $M := \{1,2,3\}$

$$M := \{x \mid x = 1 \vee x = 2 \vee x = 3\}$$

Die leere Menge wird dargestellt als:

$$\emptyset := \{\}$$

Die Reihenfolge innerhalb einer Menge ist nicht relevant.

Vordefinierte Mengen

/ : Menge der natürlichen Zahlen
$\mathbb{R}$: Menge der reellen Zahlen
$\mathbb{Z}$: Menge der ganzen Zahlen
$\mathbb{Q}$: Menge der rationalen Zahlen

Auswahl Axiom

Ist M eine Menge und ist P(x) eine Eigenschaft, die Elemente aus M halten können, dann ist:

$$N := \{x \in M \mid p(x)\}$$

Bsp.: Die Menge der geraden Zahlen:

$$M := \{x \in \mathbb{N} \mid \exists y \in \mathbb{N} : x = 2 \cdot y\}$$

Potenzmenge

Es seien M und N Mengen, dann ist M eine Teilmenge von N, geschrieben $M \subseteq N$, genau dann, wenn jedes Element von M auch ein Element von N ist.

$$M \subseteq N \Leftrightarrow \forall x : \{x \in M \rightarrow y \in N\}$$

Die Potenzmenge ist die Menge M ist die Menge aller Teilmengen von 2^M

$$2^M := \{x \mid x \subseteq M\}$$

Vereinigung von Mengen

$M \cup N := \{x \mid x \in M \vee x \in N\}$

Bsp.: $\{1,2,3\} \cup \{2,5\} = \{1,2,3,5\}$

Es ist nicht relevant, ob ein Element doppelt vorkommt.

Ist X eine Menge von Mengen, so definieren wir die Vereinigung von X als:

$\cup X := \{y \mid \exists x \in X : y \in x\}$

Bsp.: $X := \{\emptyset, \{1,2\}, \{1,3,5\}, \{7,4\}\}$

$\cup X = \{1,2,3,5,7,4\}$

Schnitt zweier Mengen

$M \cap N := \{x \mid x \in M \wedge x \in N\}$

Bsp.: $N = \{1,3,5\}$ und $M = \{2,3,5,6\}$

$\rightarrow M \cap N = \{3,5\}$

Ist X eine Menge von Mengen, so definieren wir den Schnitt von X als:

$\cap X := \{y \mid \exists x \in X : y \in x\}$

Differenzmenge

Sind M und N Mengen, so ist die Differenzmenge M (lese: M ohne N) die Menge aller Elemente aus M, die <u>nicht</u> in N enthalten sind:

Bsp.: $\{1,3,5,7\} \backslash \{2,3,5,6\} = \{1,7\}$

Bild-Menge

Es sei M eine Menge und f sei eine Funktion, die für alle x aus M definiert ist. Dann ist das Bild von M unter f definiert als:

$f(M) := \{f(x) \mid x \in M\}$

$= \{y \mid \exists x : (x \in M \wedge y = f(x))\}$

Bsp.: $f(x) = x^2, M := \{1,3,7\}$

$f(M) = \{1,9,49\}$

Kartesisches Produkt

As geordnete Paar der Objekte x und y wird geschrieben als <x, y>

X ist die erste Komponente von <x, y>

Y ist die zweite Komponente von <x, y>

Sind M und N zwei Mengen, so ist das kartesische Produkt von M und N definiert als:

$M \times N := \{ < x, y > \mid x \in M \wedge y \in N\}$

Sind $x_1, \ldots, x_n$ Objekte, so schreiben wir das sogenannte n-Tupel dieser Objekte als:

$< n_1, \ldots, x_n >$

Sind M und N Mengen, wird das allgemeine kartesische Produkt dieser Mengen wie folgt definiert:

$M_1 \times M_2 \times M_n := \{ < x_1, x_2, \ldots, x_n > \mid x_1 \in M \wedge \ldots \wedge x_n \in M_n\}$

Gleichheit von Mengen

Extensionalitäts – Axiom: Zwei Mengen A und B sind genau dann gleich, wenn sie dieselben Elemente halten.

$A = B \Leftrightarrow \forall x : (x \in A \leftrightarrow x \in B)$

Die Programmiersprache Python

Mengen in Python

In Python sind Mengen (sets) native Datentypen.
Das die leere Menge wird mit M = set() definiert
Seien A und B Mengen in Python.
Die Vereinigung von Mengen in Python:

$A \cup B =$ `A.union(B) = A | B`

Die Schnittmenge von Mengen in Python:

$A \cap B$ = A.intersection(B) = A & B

Prüfen, ob $A \subseteq B$ ist:

A <= B

Prüfen, ob $x \in M$:

x in M oder x not in M

Die Potenzmenge ($2^M := \{x \,|\, x \subseteq M\}$) einer Menge bilden:

```python
def power(M):
    if M == set():
        return { frozenset() }
    else:
         C = set(M)
         x = C.pop()
         P1 = power(C)
         P2 = {A.union({x}) for A in P1}
         return P1 | P2
```

Alle Teilmengen bestimmen in Python:

```python
def subsets(M, k):
    if k == 0:
         return { frozenset() }
    return { k | {x} for k in subsets(M, k-1)
                 for x in M
                 if x not in k}
```

Paare und das kartesische Produkt

Geordnete Paare <x, y> können in Python durch die Form (x, y) oder auch x, y dargestellt werden.

Das kartesische Produkt $A \times B$ von zwei Mengen kann wie folgt dargestellt werden:

```python
def product(A, B):
    return {(x, y) for x in A for x in B}
```

Tupel in Python

Tupel wie <1, 2, 3> können in Python durch die Form (1, 2, 3) dargestellt werden. Man kann längere Tupel beispielsweise durch ‘tupel(range(1,100+1)‘ erzeugen. Durch ‚+‘ lassen sich Tupel konkardinieren. Über Operationen wie tupelname[indexWert], lässt sich der Wert des Tupels an dem Index ausgeben.

Listen in Python

Listen verhalten sich wie Tupel, außer dass man bei ihnen die Werte an den Indizes nachträglich ändern kann.

Dictionaries in Python

Eine binäre Relation R ist eine Teilmenge des kartesischen Produkts der zwei Mengen A & B:

$$R \subseteq A \times B$$

Eine binäre Relation $R \subseteq A \times B$ ist eine funktionale Relation, wenn und nur wenn:

$$\forall x \in A : \forall y_1, y_2 \in B : (< x, y_1 > \in R \wedge < x, y_2 > \in R \rightarrow y_1 = y_2)$$

In Python kann man diese funktionalen Relationen als Dictionaries darstellen.
Bsp.:

```
In [3]: Number2English = { 1:'one', 2:'two', 3:'three', 4:'four', 5:'five',
                           6:'six', 7:'seven', 8:'eight', 9:'nine'
                         }
        Number2English

Out[3]: {1: 'one',
         2: 'two',
         3: 'three',
         4: 'four',
         5: 'five',
         6: 'six',
         7: 'seven',
         8: 'eight',
         9: 'nine'}
```

Ein Dictionary lässt sich wie folgt invertieren:

InvertiertesDictionary = {DictionaryOriginal [x]: x for x in DictionaryOriginal}

Applikationen und Fallstudien

Äquivalenzen via Fixpunktiteration lösen

Eine Fixpunktiteration ist in der Mathematik ein numerisches Verfahren zur näherungsweisen Bestimmung von Lösungen einer Gleichung oder eines Gleichungssystems.

Transitiver Abschluss

Der transitive Abschluss zweier Relationen Q & R sind binäre Relationen, dann ist das relationale Produkt definiert als:

$$Q \circ R = \{ < x, z > \mid \exists y : (< x, y > \in Q \wedge < y, z > \in R)\}$$

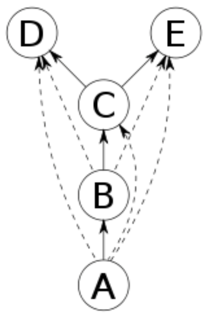
Der transitive Abschluss R^{+} binärer Relationen R ist die kleinste Relation T die gilt:

- R ist ein Subset von T $R \subseteq T$
- T ist transitiv

```python
def product( R1, R2 ):
    return { (x,z) for (x,y1) in R1 for (y2,z) in R2 if y1 == y2 }

def transitiveClosure(R):
    T = R
    while True:
        oldT = T
        T = product(R, T) | R
        if T == oldT:
            return T

```



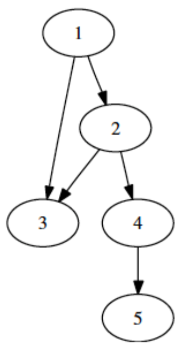
Pfade in Python

In Python werden Graphen in binären Tupeln dargestellt. Die Knoten und Kanten eines Graphen können so repräsentiert werden. Dies ermöglicht die Suche nach einem Bestimmten Pfad innerhalb des Graphen.

Bsp.:

$$R = \{(1,2), (2,3), (1,3), (2,4), (4,5)\}$$

Man berechnet den transitiven Abschluss wie bereits beschrieben.




```python
def findPaths(R):
    P = R;
    while True:
        oldP = P
        P = R | pathProduct(P, R)
        if P == oldP:
            return P

 def pathProduct(P, Q):
    return { join(S, T) for S in P for T in Q if S[-1] == T[0] }

 def join(S, T):
    return S + T[1:]
```

Grenzen der Berechenbarkeit

Das Halteproblem

Das Halteproblem befasst sich mit dem Problem, dass ein Computerprogramm nicht prüfen kann, ob ein anderes Programm terminiert $\downarrow$ oder divergiert $\uparrow$.
Ein String **t** ist eine Testfunktion mit dem Namen **f**, genau dann, wenn es die folgende Form hat.

```
"""
def f(x):
    body
"""
```

Zudem kann der String t in eine Python Methode geparst werden, sodass exec(t) keinen Fehler liefert. Die Menge aller Testfunktionen wird mit TF bezeichnet. Wenn also $t \in TF$ ist und t den Namen f hat, dann wir dies wie folgt formuliert:

$$name(t) = f$$

Das Halteproblem für Python-Funktionen ist die Frage, ob eine Python Methode stops(t, a): existiert, welche den Input einer Testfunktion t bekommt, sowie einen String a.

```
def stops(t, a):
    .
    .
    .
```

Die Stops Methode muss folgenden Spezifikationen genügen:

1. $t \notin TF \Leftrightarrow stops(t,a) \rightsquigarrow 2$
   Wenn t keine Testfunktion ist, dann gibt stops(t, a) eine 2 zurück.
2. $t \in TF \wedge name(t) = n \wedge n(a) \downarrow \Leftrightarrow stops(t,a) \rightsquigarrow 1$
   Wenn t eine Testfunktion ist und n(a) terminiert, dann gibt stops(t, a) eine 1 zurück.
3. $t \in TF \wedge name(t) = n \wedge n(a) \uparrow \Leftrightarrow stops(t,a) \rightsquigarrow 0$
   Wenn t eine Testfunktion ist und n(a) divergiert, dann gibt stops(t, a) eine 0 zurück.

Wenn es eine Python Funktion gäbe, welche die Spezifikationen erfüllt, dann wäre das Halteproblem lösbar, allerdings besagt das Theorem von Alan Turing (1936), dass das Halteproblem nicht lösbar ist.

Die Abzählbarkeit von Mengen

Eine Menge M ist abzählbar, genau dann, wenn eine Funktion $f : \mathbb{N} \rightarrow M$, die Surjektiv ist, d.h. $\forall x \in M : \exists n \in \mathbb{N} : f(n) = x$

Das Äquivalenzproblem

Das Äquivalenzproblem soll Aussage darüber liefern, ob zwei Funktionen immer das gleiche Resultat liefern.

Partielle Äquivalenz

Zwei Funktionen $n_1(a_1, ..., a_k)$ und $n_2(a_1, ..., a_k)$ sind partiell äquivalent ($\simeq$), wenn folgende Spezifikationen gelten:

1. $n_1(a_1, ..., a_k) \uparrow \wedge n_2(a_1, ..., a_k) \uparrow$
   Beide Funktionen divergieren
2. $\exists r : (n_1(a_1, ... a_k) \rightsquigarrow r \wedge n_2(a_1, ..., a_k) \rightsquigarrow r)$
   Beide Funktionen terminieren

Wenn $n_1(a_1, ..., a_k) \simeq n_2(a_1, ..., a_k)$ gilt, dann sind die Ausdrücke $n_1(a_1, ..., a_k)$ und $n_2(a_1, ..., a_k)$ partiell äquivalent.

Lösung des Äquivalenzproblems

Eine Python Funktion equals löst das Äquivalenzproblem, wenn sie wie folgt aufgebaut ist:

```
def equal(p1, p2, a):
    body
```

Und zudem noch die folgenden Spezifikationen erfüllt:

1. $p_1 \notin TF \vee p_2 \notin TF \Leftrightarrow equal(p_1, p_2, a) \rightsquigarrow 2$
2. Wenn
   a. $p_1 \in TF \wedge name(p_1) = n_1$,
   b. $p_2 \in TF \wedge name(p_2) = n_2$ und
   c. $n_1(a) \simeq n_2(a)$

   Dann muss gelten, dass

   $$equals(p_1, p_2, a) \rightsquigarrow 1$$
3. Andererseits gilt:

   $$equals(p_1, p_2, a) \rightsquigarrow 0$$

Allerdings gilt für das Äquivalenzproblem genau wie für das Halteproblem, dass es nicht lösbar ist.

Generell

Es gibt keine Prozedur, welche entscheiden kann, ob ein Programm unter einem bestimmten Input terminiert, jedoch gibt es Programme, welche dies näherungsweise versuchen.

Aussagenlogik

Überblick

Die Aussagenlogik beschäftigt sich mit der Verknüpfung einfacher Aussagen durch Junktoren. Dabei sind Junktoren wie „*und*“, „*oder*“, „*nicht*“, „*wenn …, dann*“ und „*genau dann, wenn*“. Atomare Aussagen sind Aussagen, die sich nicht weiter in Teilaussagen zerlegen lassen. Atomare Aussagen lassen sich mithilfe von Junktoren zu zusammengesetzten Aussagen verknüpfen. Aussage-Variablen sind Namen, die für atomare Aussagen stehen und des Weiteren werden in der Aussagenlogik die Junktoren „¬“, „∨“, „∧“, „→“ und „↔“ verwendet, um so Aussagenlogische Formeln beliebiger Komplexität zu erzeugen.
Aussage-Variablen verknüpft mit Junktoren sind also Aussagenlogische Formeln.
Eine aussagenlogische Formel, die immer wahr ist, wird als Tautologie bezeichnet. Eine Formel heißt erfüllbar, wenn es wenigstens eine Möglichkeit gibt, sodass die Formel wahr wird.

Formale Definition der aussagenlogischen Formeln

Die Syntax der Aussagenlogik gibt an, wie Formeln geschrieben werden und wie sich Formel zu Beweisen verknüpft lassen. Die Semantik befasst sich mit der Bedeutung der Formeln.

Syntax der Aussagenlogischen Formeln

Die Menge P von Aussage-Variablen sei gegeben. Diese besteht meist aus kleinen lateinischen Buchstaben, die zusätzlich noch indiziert werden dürfen.

$$P := \{p, q, r, p_1, p_2, p_3\}$$

Aussagenlogische Formeln sind dann Wörter, die aus dem Alphabet:

$$A := P \cup \{\top, \bot, \neg, \vee, \wedge, \rightarrow, \leftrightarrow, (,)\}$$

Menge der Aussagenlogische Formeln ***F*** Definition:

1. $\top \in F \text{und} \bot \in F$
   $\top$ steht für die Formel, die immer wahr ist und $\bot$ für die, die immer falsch ist.

2. Ist $p \in P$, so gilt auch $p \in F$
   Jede aussagenlogische Variable ist auch eine aussagenlogische Formel.

3. Ist $f \in F$, so gilt auch $\neg f \in F$
   Die Formel $\neg f$ ist die Negation der Formel f

4. Sind $f_1, f_2 \in F$, so gilt auch
   a. $(f_1 \vee f_2) \in F$
   b. $(f_1 \wedge f_2) \in F$
   c. $(f_1 \rightarrow f_2) \in F$
   d. $(f_1 \leftrightarrow f_2) \in F$

Die Operatoren „∨“, „∧“ sind links-assoziativ, wohingegen „→“ rechts-assoziativ geklammert wird. Der Operator „↔“ ist undefiniert und muss daher immer geklammert werden und durch „∧“ verknüpft werden. $p \leftrightarrow p \leftrightarrow q$ ist unzulässig und wird als $(p \leftrightarrow q) \wedge (p \leftrightarrow q)$ geschrieben.
In absteigender Bindungsstärke sind die Junktoren so aufgestellt:

„¬“ bindet am stärksten
„∨, ∧“ binden gleichstark, aber nicht so stark wie das „¬“
„→“ bindet nicht so stark wie „∨, ∧“ und
„↔“ bindet am schwächsten

Semantik der aussagenlogischen Formel

Die Semantik beschreibt die Bedeutung einer aussagenlogischen Formel. Man legt mit ihr die Interpretation oder auch die Bedeutung dieser Formel fest. Dazu werden den aussagenlogischen Formeln Wahrheitswerte zugewiesen.
Dazu wird die Menge B der Wahrheitswerte verwendet:

$$\mathbb{B} := \{True, False\}$$

Mit dieser kann nun die aussagenlogische Interpretation festgelegt werden. Eine aussagenlogische Interpretation ist eine Funktion

$$I : P \rightarrow \mathbb{B}$$

die jeder Aussage-Variablen $p \in P$ einen Wahrheitswert $I(p) \in \mathbb{B}$ zuordnet.
Eine aussagenlogische Interpretation I interpretiert die Aussage - Variablen. Die Formale Definition der aussagenlogischen Formeln sieht wie folgt aus:

1. $\widehat{\mathcal{I}}(\bot) :=$ False.
2. $\widehat{\mathcal{I}}(\top) :=$ True.
3. $\widehat{\mathcal{I}}(p) := \mathcal{I}(p)$ für alle $p \in \mathcal{P}$.
4. $\widehat{\mathcal{I}}(\neg f) := \ominus(\widehat{\mathcal{I}}(f))$ für alle $f \in \mathcal{F}$.
5. $\widehat{\mathcal{I}}(f \wedge g) := \owedge(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
6. $\widehat{\mathcal{I}}(f \vee g) := \ovee(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
7. $\widehat{\mathcal{I}}(f \rightarrow g) := \ominus\!\!\!\rightarrow(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.
8. $\widehat{\mathcal{I}}(f \leftrightarrow g) := \ominus\!\!\!\leftrightarrow(\widehat{\mathcal{I}}(f), \widehat{\mathcal{I}}(g))$ für alle $f, g \in \mathcal{F}$.

Implementierung in Python

In Python werden zusammengesetzte Datenstrukturen als geschachtelte Tupel dargestellt. Die formale Definition der Repräsentation von aussagenlogischen Formeln sieht formal wie folgt aus:

$$rep : F \rightarrow Pyton$$

Diese Funktion ordnet einer aussagenlogischen Formel f ein geschachteltes Tupel $rep(f)$ zu.

1. $rep(\top) := ('\top',)$
2. $rep(\bot) := ('\bot',)$
3. $rep(p) := p$ für alle $p \in P$
4. $rep(\neg f) := ('\neg', rep(f))$
5. $rep(f \wedge g) := ('\wedge', rep(f), rep(g))$
6. $rep(f \vee g) := ('\vee', rep(f), rep(g))$
7. $rep(f \rightarrow g) := ('\rightarrow', rep(f), rep(g))$
8. $rep(f \leftrightarrow g) := ('\leftrightarrow', rep(f), rep(g))$

Nun wird eine aussagenlogische Interpretation in Python dargestellt. Eine AL Interpretation ist eine Funktion

$$I : P \rightarrow \mathbb{B}$$

von einer Menge der Aussage-Variablen P in die Meng der Wahrheitswerte $\mathbb{B}$.
Die Funktion evaluate(F, I) erwartet zwei Argumente.

1. F eine Al Formel, die als verschachteltes Tupel dargestellt wird.
2. I eine AL Interpretation, die als Menge von aussagenlogischen-Variablen dargestellt wird.

```python
def evaluate(F, I):
    "Evaluate the propositional formula F using the interpretation I"
    if isinstance(F, str): # F is a propositional variable
        return F in I
    if F[0] == '⊤': return True
    if F[0] == '⊥': return False
    if F[0] == '¬': return not evaluate(F[1], I)
    if F[0] == '∧': return evaluate(F[1], I) and evaluate(F[2], I)
    if F[0] == '∨': return evaluate(F[1], I) or evaluate(F[2], I)
    if F[0] == '→': return not evaluate(F[1], I) or evaluate(F[2], I)
    if F[0] == '↔': return evaluate(F[1], I) == evaluate(F[2], I)
```

Tautologien

Definition Tautologie

Ist f eine aussagenlogische Formel und gilt

$$I(f) = True$$

für jede aussagenlogische Interpretation I, dann ist f eine Tautologie. In diesem Fall schreibt man

$$\models f.$$

Ist eine Formel f eine Tautologie, so lässt sich f auch als allgemeingültig bezeichnen.

Definition Äquivalenz

Zwei Formeln f und g heißen äquivalent genau dann, wenn folgendes gilt:

$$\models f \leftrightarrow g$$

Es gelten folgende Äquivalenzen:

$\models \neg\bot \leftrightarrow \top$	$\models \neg\top \leftrightarrow \bot$	
$\models p \vee \neg p \leftrightarrow \top$	$\models p \wedge \neg p \leftrightarrow \bot$	Tertium-non-Datur
$\models p \vee \bot \leftrightarrow p$	$\models p \wedge \top \leftrightarrow p$	Neutrales Element
$\models p \vee \top \leftrightarrow \top$	$\models p \wedge \bot \leftrightarrow \bot$	
$\models p \wedge p \leftrightarrow p$	$\models p \vee p \leftrightarrow p$	Idempotenz
$\models p \wedge q \leftrightarrow q \wedge p$	$\models p \vee q \leftrightarrow q \vee p$	Kommutativität
$\models (p \wedge q) \wedge r \leftrightarrow p \wedge (q \wedge r)$	$\models (p \vee q) \vee r \leftrightarrow p \vee (q \vee r)$	Assoziativität
$\models \neg\neg p \leftrightarrow p$		Elimination von $\neg\neg$
$\models p \wedge (p \vee q) \leftrightarrow p$	$\models p \vee (p \wedge q) \leftrightarrow p$	Absorption
$\models p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$	$\models p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$	Distributivität
$\models \neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$	$\models \neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$	DeMorgan'sche Regeln
$\models (p \rightarrow q) \leftrightarrow \neg p \vee q$		Elimination von $\rightarrow$
$\models (p \leftrightarrow q) \leftrightarrow (\neg p \vee q) \wedge (\neg q \vee p)$		Elimination von $\leftrightarrow$

Testen der Allgemeingültigkeit in Python

Mit Hilfe eines Python Programms soll automatisch beantwortet werden, ob eine gegebene Formel f eine Tautologie ist. Dafür wird jede mögliche Belegung der Formel untersuchen und prüfen, ob die Auswertung jedes Mal den Wert True zurückgibt.
Dafür benötigt man zuerst eine Methode collectVars(f), welche die Menge der aussagenlogischen Variablen berechnet, die in einer aussagenlogischen Formel f auftreten. Diese hat folgende Spezifikationen:

1. $\texttt{collectVars}(p) = \{p\}$ für alle aussagenlogischen Variablen p.
2. $\texttt{collectVars}(\top) = \{\}$.
3. $\texttt{collectVars}(\bot) = \{\}$.
4. $\texttt{collectVars}(\neg f) := \texttt{collectVars}(f)$.
5. $\texttt{collectVars}(f \wedge g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
6. $\texttt{collectVars}(f \vee g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
7. $\texttt{collectVars}(f \rightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.
8. $\texttt{collectVars}(f \leftrightarrow g) := \texttt{collectVars}(f) \cup \texttt{collectVars}(g)$.

Implimentiert ergibt sich nun die Python-Methode collectVars(f), welche in vier Fällen die obigen Präzidenzen zusammenfasst und erfüllt.

```python
def collectVars(f):
    "Collect all propositional variables occurring in the formula f."
    if isinstance(f, str):
        return { f }
    if f[0] in ['⊤', '⊥']:
        return set()
    if f[0] == '¬':
        return collectVars(f[1])
    return collectVars(f[1]) | collectVars(f[2])
```

Da die Allgemeingültigkeit zu berechnen ist muss nun getestet werden, ob unter jeder Variablenbelegung die aussagenlogische Formel True liefert. Dafür wird nun die Methode tautology(f) implimentiert, die für gegebene aussagenlogische Formel f überprüft, ob f eine Tautologie ist.

1. Berechnung der Menge P der aussagenlogischen Variablen, die in f auftreten
2. Mit Hilfe von power(M) und allSubsets(M, k) werden nun alle Teilmengen von P bestimmt und in der Liste A gespeichert.
3. Nun wird geprüft, ob für jede Belegung I die Auswertung der Formel f den Wert True liefert.
4. Andernfalls wird die erste Belegung I zurück gegeben, für die f den Wert False hat.

```python
def tautology(f):
    "Check, whether the formula f is a tautology."
    P = collectVars(f)
    A = power.allSubsets(P)
    if { evaluate(F, I) for I in A } == { True }:
        return True
    else:
        return [I for I in A if not evaluate(F, I)][0]
```

Man kann eine Formel auch manuell auf die Allgemeingültigkeit prüfen, indem man durch genannte Äquivalenzumformungen die Fomel bis zu ‚⊤‘ vereinfacht.

Das allgmeien Vorgehen dabei ist dabei wie folgt:

1. Bikonditonal beseitigen
2. Konditional beseitigen
3. Negations-Normalform bilden (Negationen direkt an den atomaren Aussagen)
4. Ausklammern

Bsp.: $(p \to q) \to (\neg p \to q) \to q$

$$
\begin{array}{clr}
 & (p \to q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \lor q) \to (\neg p \to q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (\neg p \lor q) \to (\neg\neg p \lor q) \to q & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (\neg p \lor q) \to (p \lor q) \to q & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & \neg(\neg p \lor q) \lor ((p \lor q) \to q) & \text{(DeMorgan)} \\
\Leftrightarrow & (\neg\neg p \land \neg q) \lor ((p \lor q) \to q) & \text{(Elimination der Doppelnegation)} \\
\Leftrightarrow & (p \land \neg q) \lor ((p \lor q) \to q) & \text{(Elimination von } \to\text{)} \\
\Leftrightarrow & (p \land \neg q) \lor (\neg(p \lor q) \lor q) & \text{(DeMorgan)} \\
\Leftrightarrow & (p \land \neg q) \lor ((\neg p \land \neg q) \lor q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \land \neg q) \lor ((\neg p \lor q) \land (\neg q \lor q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (p \land \neg q) \lor ((\neg p \lor q) \land \top) & \text{(Neutrales Element)} \\
\Leftrightarrow & (p \land \neg q) \lor (\neg p \lor q) & \text{(Distributivität)} \\
\Leftrightarrow & (p \lor (\neg p \lor q)) \land (\neg q \lor (\neg p \lor q)) & \text{(Assoziativität)} \\
\Leftrightarrow & ((p \lor \neg p) \lor q) \land (\neg q \lor (\neg p \lor q)) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & (\top \lor q) \land (\neg q \lor (\neg p \lor q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \top \land (\neg q \lor (\neg p \lor q)) & \text{(Neutrales Element)} \\
\Leftrightarrow & \neg q \lor (\neg p \lor q) & \text{(Assoziativität)} \\
\Leftrightarrow & (\neg q \lor \neg p) \lor q & \text{(Kommutativität)} \\
\Leftrightarrow & (\neg p \lor \neg q) \lor q & \text{(Assoziativität)} \\
\Leftrightarrow & \neg p \lor (\neg q \lor q) & \text{(Tertium-non-Datur)} \\
\Leftrightarrow & \neg p \lor \top & \\
\Leftrightarrow & \top &
\end{array}
$$

Definition Literal

Eine Al – Formel f heißt Literal genau dann, wenn einer der folgenden Fälle gilt:

1. $f = \top \; oder \; f = \bot$
2. $f = p$, wobei p eine Al – Variable ist.
   In diesem Fall handelt es sich um eine positives Literal
3. $f = \neg p$, wobei p eine Al – Variable ist.
   In diesem Fall hendelt es sich um ein negatives Literal

Die Menge der Literale wir mit L bezeichnet.

Definition Komplement

Ist l ein Literal, so wird das Komplement von l mit $\bar{l}$ bezeichent.

1. $\overline{\top} = \bot$ und $\overline{\bot} = \top$
2. $p := \neg p$, falls $p \in P$
3. $\neg p := p$, falls $p \in P$

Allgemein gilt:

$$\models \bar{l} \leftrightarrow \neg l$$

Definiton Klausel

Eine aussagenlogische Formel K ist eine Klausel, wenn K die Form hat

$$K = l_1 \lor \ldots \lor l_r$$

hat, wobei l_i für alle i = 1, … r ein Literal ist. Eine Klausel ist eine Disjunktion von Literalen. Die Menge aller Klauseln bezeichnet man als K. Man darf diese Disjunktion von Literalen auch in einer Menge schrieben $\{l_1, \ldots, l_r\}$. Die Leere Menge <u>{} ist äquivalent zu {⊥}</u>

Definition triviale Klausel

Eine Klausel K ist trivial, wenn einer der beiden folgenden Fälle vorliegt:

1. $\top \in K$
2. Es existiert eine Variable $p \in P$, so dass sowohl $p \in K$ als auch $\neg p \in K$ gilt.
   In diesem Fall bezeichnet man p als auch $\neg p$ als komplementäre Literale.

Es gilt, dass eine Klausel K genau dann eine Tautologie ist, wenn sie trivial ist.

Definition Konjunktive Normalform

Eine Formel F ist in konjunktiver Normalform (KNF), genau dann, wenn F eine Konjunktion von Klauseln K ist, wenn also gilt:

$$F = K_1 \wedge \ldots \wedge K_n$$

wobei K_i für alle $i = 1,...,n$ Klauseln sind.
Ebenso wie Klauseln lässt sich die KNF in einer Menge darstellen, so dass aus $F = K_1 \wedge \ldots \wedge K_n$ die folgende Menge hergeleitet werden kann

$$F = \{K_1, \ldots, K_n\}$$

Bsp.: Sind p, q und r Aussage-Variable, so ist die Formel

$$(p \vee q \vee \neg r) \wedge (q \vee \neg r \vee p \vee q) \wedge (\neg r \vee p \vee \neg q)$$

in KNF. In der Mengen-Schreibweise wird daraus

$$\{\{p, q, \neg r\}, \{p, \neg q, \neg r\}\}.$$

Verfahren für die KNF

Nun folgt ein Verfahren zur Bestimmung der KNF. Damit kann dann leicht entschieden werden, ob eine F eine Tautologie ist.

1. Elimination aller Junktoren „↔" mit Hilfe der Äquivalenzen
   $(F \leftrightarrow G) \leftrightarrow (F \rightarrow G) \wedge (G \rightarrow F)$
2. Eliminieren aller vorkommender Junktoren „→" mit Hilfe der Äquivalenzen
   $(F \leftrightarrow G) \leftrightarrow \neg F \vee G$
3. Umformung in die Negations-Normalform
   a. $\neg \bot \leftrightarrow \top$
   b. $\neg \top \leftrightarrow \bot$
   c. $\neg\neg F \leftrightarrow F$
   d. $\neg(F \wedge G) \leftrightarrow \neg F \vee \neg G$
   e. $\neg(F \vee G) \leftrightarrow \neg F \wedge \neg G$
4. Ausmultiplizieren von „∧", „∨" unter Verwendung der folgenden Äquivalenzen:
   $(F_1 \wedge \ldots \wedge F_m) \vee (G1 \wedge \ldots \wedge G_n)$
   $\rightarrow (F_1 \vee G_1) \wedge \ldots \wedge (F_1 \vee G_n) \wedge \ldots \wedge (F_m \vee G_1) \wedge \ldots \wedge (F_m \vee G_n)$
   (Der Junktur „∨" wird nach innen geschoben)
5. Umformung in die Mengenschreibweise
   Zuerst werden alle Disjunktion aller Literale in Mengen zusammengefasst und anschließend werden diese in eine Menge von Mengen zusammengefasst.

KNF – Verfahren in Python implementiert

Die einzelnen Schritte von oben werden genauso in Python implementiert.

Eliminierung des Bikonditionals:

```python
def elimBiconditional(f):
    "Eliminate the logical operator '↔' from the formula f."
    if isinstance(f, str): # This case covers variables.
        return f
    if f[0] == '↔':
        g, h = f[1:]
        ge = elimBiconditional(g)
        he = elimBiconditional(h)
        return ('∧', ('→', ge, he), ('→', he, ge))
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '¬':
        g = f[1]
        ge = elimBiconditional(g)
        return ('¬', ge)
    else:
        op, g, h = f
        ge = elimBiconditional(g)
        he = elimBiconditional(h)
        return (op, ge, he)
```

Eliminierung des Konditionals:

```python
def elimConditional(f):
    "Eliminate the logical operator '→' from f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '→':
        g, h = f[1:]
        ge = elimConditional(g)
        he = elimConditional(h)
        return ('∨', ('¬', ge), he)
    if f[0] == '¬':
        g = f[1]
        ge = elimConditional(g)
        return ('¬', ge)
    else:
        op, g, h = f
        ge = elimConditional(g)
        he = elimConditional(h)
        return (op, ge, he)
```

Negations – Normalform:

```python
def nnf(f):
    "Compute the negation normal form of f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '¬':
        g = f[1]
        return neg(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∧', nnf(g), nnf(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∨', nnf(g), nnf(h))

def neg(f):
    "Compute the negation normal form of ¬f."
    if isinstance(f, str):
        return ('¬', f)
    if f[0] == '⊤':
        return ('⊥',)
    if f[0] == '⊥':
        return ('⊤')
    if f[0] == '¬':
        g = f[1]
        return nnf(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∨', neg(g), neg(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∧', neg(g), neg(h))
```

Umwandlung in die konjunktive Normalform:

```python
def cnf(f):
    if isinstance(f, str):
        return { frozenset({f}) }
    if f[0] == '⊤':
        return set()
    if f[0] == '⊥':
        return { frozenset() }
    if f[0] == '¬':
        return { frozenset({f}) }
    if f[0] == '∧':
        g, h = f[1:]
        return cnf(g) | cnf(h)
    if f[0] == '∨':
        g, h = f[1:]
        return { k1 | k2 for k1 in cnf(g) for k2 in cnf(h) }
```

Der Herleitung - Begriff

Ist $\{f_1, \ldots, f_n\}$ eine Menge von Formeln, und g eine weitere Formel, so können wir uns fragen, ob die Formel g aus $f_1, \ldots, f_n$ folgt, ob also

$$\models f_1 \wedge \ldots \wedge f_n \rightarrow g$$

gilt. Man könnte nun das Verfahren von oben anwenden und die Formel in die konjunktive Normalform überführen, so dass aus $f_1 \wedge \ldots \wedge f_n \rightarrow g$ die Menge $\{k_1, \ldots, k_m\}$ hergeleitet wird, deren Klauseln zu der Formel $f_1 \wedge \ldots \wedge f_n \rightarrow g$ äquivalent sind. Die Formel ist nun genau dann eine Tautologie, wenn jede der Klauseln $k_1, \ldots, k_m$ trivial ist.
Ein weiteres Verfahren ist es die Schluss-Regel zu nutzen, um aus den Formeln herzuleiten.

Definition Schluss-Regel

Eine aussagenlogische Schluss-Regel ist ein Paar der Form $\langle\langle f_1, f_2\rangle, k\rangle$. Dabei ist $\langle f_1, f_2\rangle$ ein Paar von aussagenlogischen Formeln und k ist eine einzelne aussagenlogische Formel. Die beiden Formeln f_1 und f_2 bezeichnet man dabei als Prämissen, die Formel k heißt Konklusion der Schluss-Regel. Ist das Paar $\langle\langle f_1, f_2\rangle, k\rangle$ eine Schluss-Regel, schreibt man:

$$\frac{f_1 \quad f_2}{k}$$

(Gelesen: Aus f_1 und f_2 kann auf k geschlossen werden.)

Beispiele für Schluss-Regeln:

Modus Ponens	Modus Tollens	Unfug
$\dfrac{f \quad f \rightarrow g}{g}$	$\dfrac{\neg g \quad f \rightarrow g}{\neg f}$	$\dfrac{\neg f \quad f \rightarrow g}{\neg g}$

Definition der korrekten Schluss – Regel

Eine Schluss – Regel der Form $\dfrac{f_1 \quad f_2}{k}$ ist genau dann korrekt, wenn $\models f_1 \wedge f_2 \rightarrow k$ gilt.

Im Folgenden wird davon ausgegangen, dass alle Formeln Klauseln sind. Dies ist keine Einschränkung, da sich jede Formel in eine äquivalente Menge von Klauseln umformen lässt.

Definition Schnitt – Regel

Ist p eine aussagenlogische Variable und sind k_1 und k_2 Mengen von Literalen, die wir als Klauseln interpretieren, so bezeichnen wir die folgende Schluss – Regel als eine Schnitt – Regel:

$$\frac{k_1 \cup \{\neg p\} \quad \{\neg p\} \cup k_2}{\{\} \cup \{q\}}$$

Definition Herleitungs - Begriff

M sei eine Menge von Klauseln und f eine einzelne Klausel. Die Formeln aus M bezeichnen wie als Prämisse, die Formel f heißt Konklusion.

$$M \vdash f$$

Man liest „$M \vdash f$“ als „M leitet f her“.

1. Aus der Menge M von Annahmen kann jede der Annahmen hergeleitet werden:
   Falls $f \in M$ ist, dann gilt $M \vdash f$
2. Sind $k_1 \cup \{p\}$ und $\{\neg p\} \cup k_2$ Klauseln, die aus M hergeleitet werden können, so kann mit der Schnitt – Regel auch die Klausel $k_1 \cup k_2$ aus der M hergeleitet werden.
   Falls sowohl $M \vdash k_1 \cup \{p\}$ als auch $M \vdash \{\neg p\} \cup k_2$ gilt, dann gilt auch $M \vdash k_1 \cup k_2$

Bsp.: Gegeben sei die $\{\{\neg p, q\}, \{\neg q, \neg p\}\}$:

1. $\{\neg p, q\}, \{\neg q, \neg p\} \vdash \{\neg p\}$
2. $\{\neg p, q\}, \{\neg p\} \vdash \{p\}$
3. $\{\neg p\}, \{p\} \vdash \{\}$

Implementierung in Python

```python
def complement(l):
    "Compute the complement of the literal l."
    if isinstance(l, str):         # l is a propositional variable
        return ('¬', l)
    else:                          # l = ('¬', 'p')
        return l[1]

def extractVariable(l):
    "Extract the variable of the literal l."
    if isinstance(l, str):         # l is a propositional variable
        return l
    else:                           # l = ('¬', 'p')
        return l[1]

def collectVariables(M):
    "Return the set of all variables occurring in M."
    return { extractVariable(l) for C in M
                                for l in C
                }

def cutRule(C1, C2):
    '''
    Return the set of all clauses that can be deduced with the cut rule
    from the clauses c1 and c2.
    '''
    return { C1 - {l} | C2 - {complement(l)} for l in C1
                                          if complement(l) in C2
    }
```

Die Funktion cutRule erhält als Argumente zwei Klauseln C1 und C2 und berechnet die Menge aller Klauseln, die mit Hilfe einer Anwendung der Schnitt-Regel aus C1 und C2 gefolgert werden können. Beispielsweise können wir aus den beiden Klauseln

$\{p, q\}$ und $\{\neg p, \neg q\}$

mit der Schnitt – Regel sowohl die Klausel

$\{q, \neg q\}$ als auch die Klausel $\{p, \neg q\}$

Herleiten.

```
def saturate(Clauses):
    while True:
        Derived = { C for C1 in Clauses
                              for C2 in Clauses
                              for C in cutRule(C1, C2)
                         }
        if frozenset() in Derived:
            return { frozenset() }       # This is the set notation of ⊥.
        Derived -= Clauses
        if Derived == set():             # no new clauses found
            return Clauses
        Clauses |= Derived
```

Diese Funktion erhält als Eingabe eine Menge Clauses von aussagenlogischen Klauseln, die als Mengen von Literalen dargestellt werden. Aufgabe der Funktion ist es, alle Klauseln herzuleiten, die mit Hilfe der Schnitt-Regel auf direktem oder indirektem Weg aus der Menge Clauses hergeleitet werden können. Genauer sagen wir, dass die Menge S der Klauseln, die von der Funktion saturate zurückgegeben wird, unter Anwendung der Schnitt-Regel saturiert ist, was formal wie folgt definiert ist:

1. Falls S die leere Klausel {} enthält, dann ist S saturiert.
2. Andernfalls muss Clauses eine Teilmenge von S sein und es muss zusätzlich Folgendes gelten:
Falls für ein Literal l sowohl die Klausel $C_1 \cup \{l\}$ als auch die Klausel $C_2 \cup \{\bar{l}\}$ Klausel in S enthalten ist, dann ist auch die Klausel $C_1 \cup C_2$ ein Element der Klauselmenge S:

$$C_1 \cup \{l\} \in S \wedge C_2 \cup \{\bar{l}\} \in S \Rightarrow C_1 \cup C_2 \in S$$

findValuation erhält als Eingabe eine Menge Clauses von Klauseln. Falls diese Menge widersprüchlich ist, soll die Funktion das Ergebnis False zurückgeben. Andernfalls soll eine aussagenlogische Belegung I berechnet werden, unter der alle Klauseln aus der Menge Clauses erfüllt sind. Im Detail arbeitet die Funktion findValuation wie folgt:

```
def findValuation(Clauses):
    "Given a set of Clauses, find an interpretation satisfying all clauses."
    Variables = collectVariables(Clauses)
    Clauses = saturate(Clauses)
    if frozenset() in Clauses: # The set Clauses is inconsistent.
        return False
    Literals = set()
    for p in Variables:
        if any(C for C in Clauses
                       if p in C and C - {p} <= { complement(l) for l in Literals }

                  ):
            Literals |= { p }
        else:
            Literals |= { ('¬', p) }
     return Literals
```

Das Verfahren von Davis und Putnam

Das Verfahren von Davis und Putnam entscheidet über die Erfüllbarkeit einer aussagenlogischen Formel in konjunktiver Normalform

In der Praxis stellt sich oft die Aufgabe, für eine Menge von Klauseln K eine aussagenlogische Belegung I zu berechnen, so dass

$$evaluate(C, I) = True \text{ für alle } C \in K$$

gilt. In diesem Fall kann man sagen, dass die Belegung I eine Lösung der Klause – Menge K ist.
Zur Wiederholung: Eine Klauselmenge ist eine Konjunktion von Literalen.

Betrachte man also die drei folgenden Beispiele:

$$K_1 = \{ \{p\}, \{\neg q\}, \{r\}, \{\neg s\}, \{\neg t\} \}$$

Die Klausel-Menge K_1 entspricht der aussagenlogischen Formel

$$p \wedge \neg q \wedge r \wedge \neg s \wedge \neg t.$$

Daher ist K_1 lösbar und die Belegung

$$\mathcal{I} = \{ \langle p, \texttt{True} \rangle, \langle q, \texttt{False} \rangle, \langle r, \texttt{True} \rangle, \langle s, \texttt{False} \rangle, \langle t, \texttt{False} \rangle \}$$

ist eine Lösung. Betrachten wir eine weiteres Beispiel:

$$K_2 = \{ \{\}, \{p\}, \{\neg q\}, \{r\} \}$$

Diese Klausel-Menge entspricht der Formel

$$\bot \wedge p \wedge \neg q \wedge r.$$

Offensichtlich ist K_2 unlösbar. Als letztes Beispiel betrachten wir

$$K_3 = \{\{p\}, \{\neg q\}, \{\neg p\}\}.$$

Diese Klausel-Menge kodiert die Formel

$$p \wedge \neg q \wedge \neg p$$

und ist offenbar ebenfalls unlösbar, denn eine Lösung $\mathcal{I}$ müsste die aussagenlogische Variable p gleichzeitig wahr und falsch machen. Wir nehmen die an den letzten drei Beispielen gemachten Beobachtungen zum Anlass für zwei Definitionen.

Definition Unit – Klausel

Eine Klausel C heißt Unit – Klausel, wenn C aus nur einem Literal besteht. Es gilt dann entweder

$$C = \{p\} \text{ oder } C = \{\neg p\}$$

Für eine geeignete Aussage – Variable p.

Definition triviale Klausel – Menge

Eine Klausel – Menge K heißt trivial, wenn einer der beiden folgenden Fälle vorliegt:

1. K enthält die leere Klausel (entspricht Falsum), es gilt also $\{\} \in K$
   In diesem Fall ist K offensichtlich <u>unlösbar</u>
2. K enthält eine Unit – Klausel mit <u>verschiedenen</u> Aussage – Variablen. D.h. es kann nicht sein, dass es eine aussagenlogische Variable p gibt, so dass K sowohl die Klausel $\{p\}$, als auch die Klausel $\{\neg p\}$ enthält. Somit gilt de facto der folgende Ausdruck:

$$(\forall C \in K : card(C) = 1) \wedge \forall p \in P : \neg(\{p\} \in K \wedge \{\neg p\} \in K)$$

In diesem Fall ist K ebenfalls unlösbar und man kann die aussagenlogische Belegung I wie folgt definieren:

$$\mathcal{I}(p) = \begin{cases} \texttt{True} & \text{falls} \quad \{p\} \in K, \\ \texttt{False} & \text{falls} \quad \{\neg p\} \in K. \end{cases}$$

Dann ist I eine Lösung der Klausel – Menge K.
Es gibt drei Möglichkeiten eine Menge von Klauseln so zu vereinfachen, dass sie nur noch aus Unit – Klauseln besteht:

1. Schnitt – Regel
2. Subsumption und
3. Fallunterscheidung

Vereinfachung mit der Schnitt – Regel

Eine typische Anwendung der Schnitt – Regel hat die Form:

$$\frac{C_1 \cup \{p\} \quad \{\neg p\} \cup C_2}{C_1 \cup C_2}$$

In diesem Verfahren lassen wir die Schnitt - Regel nur zu, wenn einer der beiden Klauseln eine Unit – Klausel ist. Dann handelt es sich um sogenannte Unit – Schnitte. Ein Unit – Schnitt oder Unit – Cut hat dann die Form:

$$unitCut(K, l) = \{C \backslash \{\bar{l}\} \mid C \in K\}$$

Man darf den Unit – Cut nur ausführen, wenn die Klausel $\{l\}$ ein Element der Menge K ist.

EINFACH AUSGEDRÜCKT:
Beim Unit – Cut werden alle Komplimente des Laterals p aus den Klauseln gestrichen und das Literal $\{p\}$ in die Menge von Klauseln angefügt, bei der Fallunterscheidung (folgt).
Bsp.: $K := \{\{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \{\neg p, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}, \{p\}\}$ sei eine gegebene Klauselmenge, dann ist nach dem Schnitt mit $\{p\}$ folgende Menge über:

$$= \{\{r, \neg t\}, \{r, s\}, \{\neg r, q\}, \{, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \underline{\underline{\{\neg s\}}}, \{p\}$$

Als nächstes würde man dann mit $\{\neg s\}$ schneiden, da diese Klausel eine neu erzeugte Unit – Klausel ist. Man führt den Prozess so lange durch, bis man eine triviale Klauselmenge erhält oder den Nachweis, dass sie nicht lösbar ist, genau dann, wenn $\{\} \in K$.

Vereinfachung durch Subsumtion

Das Prinzip der Subsumtion wird an folgendem Beispiel klar. Zunächst wird

$$K = \{\{p, q, \neg r\}, \{p\}\} \cup M$$

betrachtet. Offenbar impliziert $\{p\}$ die Klausel $\{p, q, \neg r\}$, denn immer, wenn $\{p\}$ erfüllt ist, ist automatisch auch $\{p, q, \neg r\}$ erfüllt. Das liegt daran, dass

$$\models p \rightarrow q \vee p \vee \neg r$$

Gilt. Allgemein kann man sagen, dass eine Klausel C von einer Unit – Klausel U subsumiert wird, wenn $U \subseteq C$ gilt.

EINFACH AUSGEDRÜCKT:
Bei der Subsumtion wird in der Menge von Klauseln jede Menge eliminiert, welche das Literal p selbst enthält. Auch hier wird wieder das Literal $\{p\}$ selbst mit an die Menge eingefügt, bei der Fallunterscheidung (folgt).
BSP.:

$$K := \{\{p, q, s\}, \{\neg p, r, t\}, \{r, s\}, \{\neg r, \neg p, q\}, \{\neg s, p\}, \{\neg p, \neg q, s, \neg r\}, \{p, \neg q, s\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}\}$$

Subsumtion der Klauselmenge K mit $\{p\}$ führt zu:

$$= \{\cancel{\{p, q, s\}}, \{\neg p, r, t\}, \{r, s\}, \{\neg r, \neg p, q\}, \cancel{\{\neg s, p\}}, \{\neg p, \neg q, s, \neg r\}, \cancel{\{p, \neg q, s\}}, \{\neg r, \neg s\}, \{\neg p, \neg s\}, \{p\}\}$$

$$= \{\neg p, r, t\}, \{r, s\}, \{\neg r, \neg p, q\}, \{\neg p, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}, \{p\}\}$$

Vereinfachung durch Fallunterscheidung

Da beide Methoden, Unit – Cut und Subsumtion nur funktionieren, wenn eine Unit – Klausel in der Klauselmenge enthalten ist, muss man Sorge tragen, dass es auch wirklich eine Unit – Klausel gibt. Wenn in der gegebenen Klausel – Menge K keine Unit – Klausel vorhanden ist, dann macht man an der Stelle ein Fallunterscheidung. Man nimmt am besten ein Literal, welches sich für eine Subsumtion und/ oder einen Unit – Cut anbietet und fügt es einfach zu der Klausel – Menge K hinzu. Ab besten erkennt man den Zusammenhang an einem Beispiel:

$$K := \{\{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \{\neg p, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}\}$$

sei gegeben, aber es existiert keine Unit – Klausel, also führ man eine Fallunterscheidung mit $\{p\}$ und $\{\neg p\}$ durch. Wenn nämlich eine der beiden Wege dann zu einer Lösung führt, ist dies ein gültiger Weg, um zu zeigen, dass es sich um eine triviale Klausel – Menge handelt.

Fall 1 $K \cup \{\{p\}\}$:

$$K := \{\{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \{\neg p, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}, \{p\}\}$$

Fall 2 $K \cup \{\{\neg p\}\}$:

$$K := \{\{\neg p, r, \neg t\}, \{r, s\}, \{\neg r, q, \neg p\}, \{\neg p, \neg q, s, \neg r\}, \{\neg r, \neg s\}, \{\neg p, \neg s\}, \{\neg p\}\}$$

Für beide Fälle führt man nun die Subsumtion und/ oder der Unit – Cut wie gehabt aus.

Der Algorithmus

Ziel des Algorithmus ist es eine Belegung I für die Menge K von Klauseln zu finden, so dass gilt:

$$I(C) = True \text{ für alle } C \in K$$

Das Verfahren besteht nun folgenden Schritten:

1. Alle Unit – cuts und Subsumptionen ausführen, die für die Klausel – Menge K möglich sind.
2. Falls K nun trivial ist, ist man fertig.
3. Andernfalls wählt man eine aussagenlogische Variable p, die in K auftritt.
   a. Rekursiv wird bei Schritt 1 begonnen mit folgender Klausel – Menge:
      $K \cup \{\{p\}\}$.
      Falls K nun trivial ist, kann man aufhören.
   b. Ansonsten wird Schritt 1 wieder ausgeführt, nur mit folgender Klausel – Menge:
      $K \cup \{\{\neg p\}\}$.
      Wenn dies auch fehlschlägt, dann ist K unlösbar, sonst hat man nun eine Lösung für K.

Implementierung des Algorithmus von Davis und Putnam

Für die Implementierung ist wichtig zu wissen, dass die beiden Schritte Unit – Cut und Subsumption in einer Methode zusammengefasst werden:

$$\texttt{reduce}(K,l) = \left\{ C \backslash \{\overline{l}\} \mid C \in K \wedge \overline{l} \in C \right\} \cup \left\{ C \in K \mid \overline{l} \notin C \wedge l \notin C \right\} \cup \{\{l\}\}$$

Nun folgt die Implementierung der Funktion Solve, mit der die Frage, ob eine Menge von Klauseln erfüllbar ist, beantwortet werden kann. Die Funktion erhält zwei Argumente: Die Menge der *Clauses* und der *Variables*. Clauses ist eine Menge von Klauseln und Variables ist eine Menge von Variablen. Falls die Menge Clauses erfüllbar ist, so liefert der Aufruf

$$solve(Clauses, Variables)$$

Eine Menge von Unit – Klauseln *Result*, so dass jede Belegung I, die alle Unit – Klauseln aus Result erfüllt, auch alle Klauseln aus der Menge Clauses erfüllt. Falls die Menge Clauses nicht erfüllbar ist, liefert der Aufruf

$$solve(Clauses, Variables)$$

als Ergebnis die Menge $\{\{\}\}$ zurück, denn die leere Klausel repräsentiert die unerfüllbare Formel $\perp$.

```
def solve(Clauses, Variables):
    S = saturate(Clauses);
    empty = frozenset()
    Falsum = {empty}
    if empty in S:                    # S is inconsistent
        return Falsum
    if all(len(C) == 1 for C in S):   # S is trivial
        return S
    p = selectVariable(S, Variables)
    negP = complement(p)
    Result = solve(S | { frozenset({p}) }, Variables | { p })
    if Result != Falsum:
        return Result
    return solve(S | { frozenset({negP}) }, Variables| { p })
```

```
def saturate(Clauses):
    S = Clauses.copy()
    Units = { C for C in S if len(C) == 1 }
    Used = set()
    while len(Units) > 0:
        unit = Units.pop()
        Used |= { unit }
        l = arb(unit)
        S = reduce(S, l)
        Units = { C for C in S if len(C) == 1 } - Used
    return S
```

```
def reduce(Clauses, l):
    lBar = complement(l)
    return { C - { lBar } for C in Clauses if lBar in C } \
                         | { C for C in Clauses if lBar not in C and l not in C } \
                         | { frozenset({l}) }
```

```python
def selectLiteral(Clauses, Forbidden):
    Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
    return arb(Variables)
```

```python
def arb(S):
    "Return some member from the set S."
    for x in S:
        return x
```

Codes zum Üben:

Power

```python
def power(M):
    if M == set():
        return { frozenset() }
    else:
         C = set(M)
         x = C.pop()
         P1 = power(C)
         P2 = {A.union({x}) for A in P1}
         return P1 | P2
```

Subset

```python
def subsets(M, k):
    if k == 0:
         return { frozenset() }
    return { k | {x} for k in subsets(M, k-1)
                 for x in M
                 if x not in k}
```

Product

```python
def product( R1, R2 ):
   return { (x,z) for (x,y1) in R1 for (y2,z) in R2 if y1 == y2 }

```

TrabnsitiveClosure

```python
def transitiveClosure(R):
    T = R
    while True:
        oldT = T
        T = product(R, T) | R
        if T == oldT:
            return T
```

FindPaths

```python
def findPaths(R):
    P = R;
    while True:
        oldP = P
        P = R | pathProduct(P, R)
        if P == oldP:
            return P

```

PathProduct

```python
 def pathProduct(P, Q):
    return { join(S, T) for S in P for T in Q if S[-1] == T[0] }
```

Join

```python
def join(S, T):
    return S + T[1:]

```

Evaluate

```python
def evaluate(F, I):
    "Evaluate the propositional formula F using the interpretation I"
    if isinstance(F, str): # F is a propositional variable
        return F in I
    if F[0] == '⊤': return True
    if F[0] == '⊥': return False
    if F[0] == '¬': return not evaluate(F[1], I)
    if F[0] == '∧': return evaluate(F[1], I) and evaluate(F[2], I)
    if F[0] == '∨': return evaluate(F[1], I) or evaluate(F[2], I)
    if F[0] == '→': return not evaluate(F[1], I) or evaluate(F[2], I)
    if F[0] == '↔': return evaluate(F[1], I) == evaluate(F[2], I)
```

CollectVars

```python
def collectVars(f):
    "Collect all propositional variables occurring in the formula f."
    if isinstance(f, str):
        return { f }
    if f[0] in ['⊤', '⊥']:
        return set()
    if f[0] == '¬':
        return collectVars(f[1])
    return collectVars(f[1]) | collectVars(f[2])
```

Tautology

```python
def tautology(f):
    "Check, whether the formula f is a tautology."
    P = collectVars(f)
    A = power.allSubsets(P)
    if { evaluate(F, I) for I in A } == { True }:
        return True
    else:
        return [I for I in A if not evaluate(F, I)][0]
```

ElimBiconditional

```python
def elimBiconditional(f):
    "Eliminate the logical operator '↔' from the formula f."
    if isinstance(f, str): # This case covers variables.
        return f
    if f[0] == '↔':
        g, h = f[1:]
        ge = elimBiconditional(g)
        he = elimBiconditional(h)
        return ('∧', ('→', ge, he), ('→', he, ge))
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '¬':
        g = f[1]
        ge = elimBiconditional(g)
        return ('¬', ge)
    else:
        op, g, h = f
        ge = elimBiconditional(g)
        he = elimBiconditional(h)
        return (op, ge, he)
```

ElimConditional

```python
def elimConditional(f):
    "Eliminate the logical operator '→' from f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '→':
        g, h = f[1:]
        ge = elimConditional(g)
        he = elimConditional(h)
        return ('∨', ('¬', ge), he)
    if f[0] == '¬':
        g = f[1]
        ge = elimConditional(g)
        return ('¬', ge)
    else:
        op, g, h = f
        ge = elimConditional(g)
        he = elimConditional(h)
        return (op, ge, he)
```

Nnf & Neg

```
def nnf(f):
    "Compute the negation normal form of f."
    if isinstance(f, str):
        return f
    if f[0] == '⊤':
        return f
    if f[0] == '⊥':
        return f
    if f[0] == '¬':
        g = f[1]
        return neg(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∧', nnf(g), nnf(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∨', nnf(g), nnf(h))

def neg(f):
    "Compute the negation normal form of ¬f."
    if isinstance(f, str):
        return ('¬', f)
    if f[0] == '⊤':
        return ('⊥',)
    if f[0] == '⊥':
        return ('⊤')
    if f[0] == '¬':
        g = f[1]
        return nnf(g)
    if f[0] == '∧':
        g, h = f[1:]
        return ('∨', neg(g), neg(h))
    if f[0] == '∨':
        g, h = f[1:]
        return ('∧', neg(g), neg(h))
```

Cnf

```
def cnf(f):
    if isinstance(f, str):
        return { frozenset({f}) }
    if f[0] == '⊤':
        return set()
    if f[0] == '⊥':
        return { frozenset() }
    if f[0] == '¬':
        return { frozenset({f}) }
    if f[0] == '∧':
        g, h = f[1:]
        return cnf(g) | cnf(h)
    if f[0] == '∨':
        g, h = f[1:]
        return { k1 | k2 for k1 in cnf(g) for k2 in cnf(h) }
```

Complement

```
def complement(l):
    "Compute the complement of the literal l."
    if isinstance(l, str):          # l is a propositional variable
        return ('¬', l)
    else:                           # l = ('¬', 'p')
        return l[1]
```

ExtractVariables

```
def extractVariable(l):
    "Extract the variable of the literal l."
    if isinstance(l, str):         # l is a propositional variable
        return l
    else:                           # l = ('¬', 'p')
        return l[1]

```

CollectVariables

```
def collectVariables(M):
    "Return the set of all variables occurring in M."
    return { extractVariable(l) for C in M
                                for l in C
                 }
```

CutRule

```
def cutRule(C1, C2):
    '''
    Return the set of all clauses that can be deduced with the cut rule
    from the clauses c1 and c2.
    '''
    return { C1 - {l} | C2 - {complement(l)} for l in C1
                                             if complement(l) in C2
    }
```

Saturate

```
def saturate(Clauses):
    while True:
        Derived = { C for C1 in Clauses
                          for C2 in Clauses
                          for C in cutRule(C1, C2)
                  }
        if frozenset() in Derived:
            return { frozenset() }       # This is the set notation of ⊥.
        Derived -= Clauses
        if Derived == set():             # no new clauses found
            return Clauses
        Clauses |= Derived
```

FindValuation

```
def findValuation(Clauses):
    "Given a set of Clauses, find an interpretation satisfying all clauses."
    Variables = collectVariables(Clauses)
    Clauses = saturate(Clauses)
    if frozenset() in Clauses: # The set Clauses is inconsistent.
        return False
    Literals = set()
    for p in Variables:
        if any(C for C in Clauses
                   if p in C and C - {p} <= { complement(l) for l in Literals }

              ):
            Literals |= { p }
        else:
            Literals |= { ('¬', p) }
     return Literals
```

Solve

```python
def solve(Clauses, Variables):
    S = saturate(Clauses);
    empty = frozenset()
    Falsum = {empty}
    if empty in S:                          # S is inconsistent
        return Falsum
    if all(len(C) == 1 for C in S):         # S is trivial
        return S
    p = selectVariable(S, Variables)
    negP = complement(p)
    Result = solve(S | { frozenset({p}) }, Variables | { p })
    if Result != Falsum:
        return Result
    return solve(S | { frozenset({negP}) }, Variables| { p })
```

Saturate

```python
def saturate(Clauses):
    S = Clauses.copy()
    Units = { C for C in S if len(C) == 1 }
    Used = set()
    while len(Units) > 0:
        unit = Units.pop()
        Used |= { unit }
        l = arb(unit)
        S = reduce(S, l)
        Units = { C for C in S if len(C) == 1 } - Used
    return S
```

Reduce

```python
def reduce(Clauses, l):
    lBar = complement(l)
    return { C - { lBar } for C in Clauses if lBar in C } \
                       | { C for C in Clauses if lBar not in C and l not in C } \
                       | { frozenset({l}) }
```

SelectLiteral

```python
def selectLiteral(Clauses, Forbidden):
    Variables = { extractVariable(l) for C in Clauses for l in C } - Forbidden
    return arb(Variables)
```

Arb

```python
def arb(S):
    "Return some member from the set S."
    for x in S:
         return x
```