Laporan Kecerdasan Komputasional

Nama : Alif As'ad Ramadhan

NRP : 5054231007

1. Logical Sentences

Kelas Expr dirancang untuk mewakili semua jenis ekspresi matematika. Tipe Expr yang paling sederhana adalah simbol, yang dapat didefinisikan dengan fungsi Symbol:

In [2]:
```
Symbol('x')
```

Out[2]:
x

Kita juga bisa mendefinisikan beberapa symbol sekaligus dengan fungsi Symbol

In [3]:
```
(x, y, P, Q, f) = symbols('x, y, P, Q, f')
```

Kita dapat menggabungkan Exprs dengan operator infiks dan prefiks Python biasa. Berikut ini cara kita membentuk kalimat logis "P dan bukan Q":

In [4]:
```
P & ~Q
```

Out[4]:
(P & ~Q)

`Expr` memiliki dua kolom: `op` untuk operator, yang selalu berupa string, dan `args` untuk argumen, yang merupakan tuple dari 0 atau lebih ekspresi. Yang saya maksud dengan "ekspresi" adalah contoh dari `Expr`, atau angka. Mari kita lihat kolom untuk beberapa contoh `Expr`

In [5]:
```
sentence = P & ~Q

sentence.op
```

Out[5]: '&'

In [6]:
```
sentence.args
```

Out[6]: (P, ~Q)

```
In [7]: P.op
Out[7]: 'P'
In [8]: P.args
Out[8]: ()
In [9]: Pxy = P(x, y)
        Pxy.op
Out[9]: 'P'
In [10]: Pxy.args
Out[10]: (x, y)
```

Penting untuk dicatat bahwa kelas Expr tidak mendefinisikan logika kalimat Logika Proposisional, kelas ini hanya memberi Anda cara untuk merepresentasikan ekspresi. Anggaplah Expr sebagai pohon sintaksis abstrak. Setiap argumen dalam Expr dapat berupa simbol, angka, atau Expr bersarang. Kita dapat menumpuk pohon ini hingga kedalaman berapa pun. Berikut adalah Expr bersarang yang dideploy.

In [11]:
```
3 * f(x, y) + P(y) / 2 + 1
```

Out[11]:
```
(((3 * f(x, y)) + (P(y) / 2)) + 1)
```

2. Operators for Constructing Logical Sentences

Berikut adalah tabel operator yang dapat digunakan untuk membentuk kalimat.

Operation	Book	Python Infix Input	Python Output	Python `Expr` Input
Negation	¬ P	`~P`	`~P`	`Expr('~', P)`
And	P ∧ Q	`P & Q`	`P & Q`	`Expr('&', P, Q)`
Or	P ∨ Q	`P | Q`	`P | Q`	`Expr('|', P, Q)`
Inequality (Xor)	P ≠ Q	`P ^ Q`	`P ^ Q`	`Expr('^', P, Q)`
Implication	P → Q	`P | '==>' | Q`	`P ==> Q`	`Expr('==>', P, Q)`
Reverse Implication	Q ← P	`Q | '<==' | P`	`Q <== P`	`Expr('<==', Q, P)`
Equivalence	P ↔ Q	`P | '<=>' | Q`	`P <=> Q`	`Expr('<=>', P, Q)`

Berikut ini adalah contoh pendefinisian kalimat dengan panah implikasi

In [12]:
```
~(P & Q)  |'==>'|  (~P | ~Q)
```

Out[12]:
```
(~(P & Q) ==> (~P | ~Q))
```

Jika notasi |'==>'| terlihat tidak menarik bagi Anda, Anda dapat menggunakan fungsi expr sebagai gantinya:

In [13]:
```
expr('~(P & Q)  ==>  (~P | ~Q)')
```

Out[13]:
```
(~(P & Q) ==> (~P | ~Q))
```

expr mengambil string sebagai input, dan menguraikannya menjadi Expr. String tersebut dapat berisi operator panah: ==>, <==, atau <=>, yang ditangani seolah-olah merupakan operator infiks Python biasa. Dan expr secara otomatis mendefinisikan simbol apa pun, jadi Anda tidak perlu mendefinisikannya terlebih dahulu

In [14]:
```
expr('sqrt(b ** 2 - 4 * a * c)')
```

Out[14]:
```
sqrt(((b ** 2) - ((4 * a) * c)))
```

3. Propositional Knowledge Bases: PropKB

Kelas PropKB dapat digunakan untuk merepresentasikan basis pengetahuan kalimat logika proposisional.

Kita melihat bahwa kelas KB memiliki empat metode, selain __init__. Hal yang perlu diperhatikan di sini: metode ask cukup memanggil metode ask_generator. Jadi, metode ini telah diimplementasikan, dan yang harus Anda implementasikan saat membuat kelas basis pengetahuan Anda sendiri (meskipun Anda mungkin tidak akan pernah membutuhkannya, mengingat kelas yang telah kami buat untuk Anda) adalah fungsi ask_generator dan bukan fungsi ask itu sendiri.

Class PropKB sekarang:

- `__init__(self, sentence=None)` : The constructor `__init__` creates a single field `clauses` which will be a list of all the sentences of the knowledge base. Note that each one of these sentences will be a 'clause' i.e. a sentence which is made up of only literals and `or`s.
- `tell(self, sentence)` : When you want to add a sentence to the KB, you use the `tell` method. This method takes a sentence, converts it to its CNF, extracts all the clauses, and adds all these clauses to the `clauses` field. So, you need not worry about `tell`ing only clauses to the knowledge base. You can `tell` the knowledge base a sentence in any form that you wish; converting it to CNF and adding the resulting clauses will be handled by the `tell` method.

- `ask_generator(self, query)` : The `ask_generator` function is used by the `ask` function. It calls the `tt_entails` function, which in turn returns `True` if the knowledge base entails query and `False` otherwise. The `ask_generator` itself returns an empty dict `{}` if the knowledge base entails query and `None` otherwise. This might seem a little bit weird to you. After all, it makes more sense just to return a `True` or a `False` instead of the `{}` or `None` But this is done to maintain consistency with the way things are in First-Order Logic, where an `ask_generator` function is supposed to return all the substitutions that make the query true. Hence the dict, to return all these substitutions. I will be mostly be using the `ask` function which returns a `{}` or a `False`, but if you don't like this, you can always use the `ask_if_true` function which returns a `True` or a `False`.
- `retract(self, sentence)` : This function removes all the clauses of the sentence given, from the knowledge base. Like the `tell` function, you don't have to pass clauses to remove them from the knowledge base; any sentence will do fine. The function will take care of converting that sentence to clauses and then remove those.

4. Wumpus World KB

Mari kita membuat PropKB untuk dunia wumpus dengan kalimat-kalimat yang disebutkan di atas.

In [15]:

```
wumpus_kb = PropKB()
```

We define the symbols we use in our clauses.
$P_{x,y}$ is true if there is a pit in `[x, y]`.
$B_{x,y}$ is true if the agent senses breeze in `[x, y]`.

In [16]:

```
P11, P12, P21, P22, P31, B11, B21 = expr('P11, P12, P21, P22, P31, B11, B21')
```

Sekarang kita sampaikan kalimat berdasarkan kalimat kalimat yang disebutkan di atas Tidak ada pit di [1,1]

In [17]:

```
wumpus_kb.tell(~P11)
```

.Suatu kotak dikatakan berangin jika dan hanya jika ada lubang di kotak di sebelahnya. Hal ini harus dinyatakan untuk setiap kotak, tetapi untuk saat ini, kami hanya menyertakan kotak yang relevan.

In [18]:

```
wumpus_kb.tell(B11 | '<=>' | ((P12 | P21)))
wumpus_kb.tell(B21 | '<=>' | ((P11 | P22 | P31)))
```

Sekarang kita sertakan persepsi Breeze untuk dua kotak pertama.

In [19]:

```
wumpus_kb.tell(~B11)
wumpus_kb.tell(B21)
```

Kita dapat memeriksa klausa yang disimpan dalam KB dengan mengakses variabel klausanya.

In [20]:

```
wumpus_kb.clauses
```

Out[20]:

```
[~P11,
 (~P12 | B11),
 (~P21 | B11),
 (P12 | P21 | ~B11),
 (~P11 | B21),
 (~P22 | B21),
 (~P31 | B21),
 (P11 | P22 | P31 | ~B21),
 ~B11,
 B21]
```

Berikut adalah penjelasan dari output yang didapatkan di atas.

We see that the equivalence $B_{1,1} \iff (P_{1,2} \vee P_{2,1})$ was automatically converted to two implications which were inturn converted to CNF which is stored in the `KB`.
$B_{1,1} \iff (P_{1,2} \vee P_{2,1})$ was split into $B_{1,1} \implies (P_{1,2} \vee P_{2,1})$ and $B_{1,1} \impliedby (P_{1,2} \vee P_{2,1})$.
$B_{1,1} \implies (P_{1,2} \vee P_{2,1})$ was converted to $P_{1,2} \vee P_{2,1} \vee \neg B_{1,1}$.
$B_{1,1} \impliedby (P_{1,2} \vee P_{2,1})$ was converted to $\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}$ which becomes
$(\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$ after applying De Morgan's laws and distributing the disjunction.
$B_{2,1} \iff (P_{1,1} \vee P_{2,2} \vee P_{3,2})$ is converted in similar manner.

p	q	$p \Rightarrow q$	$q \Rightarrow p$	$p \Leftrightarrow q$
F	F	T	T	T
F	T	T	F	F
T	F	F	T	F
T	T	T	T	T

5. Knowledge based agents

Agen berbasis pengetahuan adalah agen generik sederhana yang memelihara dan menangani basis pengetahuan. Basis pengetahuan tersebut mungkin awalnya berisi beberapa pengetahuan latar belakang.

Tujuan agen KB adalah untuk menyediakan tingkat abstraksi atas manipulasi basis pengetahuan dan akan digunakan sebagai kelas dasar bagi agen yang bekerja pada basis pengetahuan.

Dengan adanya persepsi, agen KB menambahkan persepsi tersebut ke basis pengetahuannya, meminta basis pengetahuan untuk melakukan tindakan terbaik, dan memberi tahu basis pengetahuan bahwa ia telah mengambil tindakan tersebut.

Implementasi KB-Agent kami dienkapsulasi dalam kelas KB_AgentProgram yang mewarisi dari kelas KB.

Mari kita lihat.

In [21]:

```
psource(KB_AgentProgram)
```

```
def KB_AgentProgram(KB):
    """A generic logical knowledge-based agent program. [Figure 7.1]"""
    steps = itertools.count()

    def program(percept):
        t = next(steps)
        KB.tell(make_percept_sentence(percept, t))
        action = KB.ask(make_action_query(t))
        KB.tell(make_action_sentence(action, t))
        return action

    def make_percept_sentence(percept, t):
        return Expr("Percept")(percept, t)

    def make_action_query(t):
        return expr("ShouldDo(action, {})".format(t))

    def make_action_sentence(action, t):
        return Expr("Did")(action[expr('action')], t)

    return program
```

Fungsi pembantu make_percept_sentence, make_action_query, dan make_action_sentence semuanya diberi nama dengan tepat dan sebagaimana yang diharapkan, make_percept_sentence membuat kalimat logika tingkat pertama tentang persepsi yang kita inginkan agar diterima agen

kita, make_action_query menanyakan KB yang mendasarinya tentang tindakan yang harus diambil, dan make_action_sentence memberi tahu KB yang mendasarinya tentang tindakan yang baru saja diambilnya.

6. Inference in Propositional Knowledge Base

Pada bagian ini kita akan melihat dua algoritma untuk memeriksa apakah sebuah kalimat mengandung KB. Tujuan kita adalah untuk memutuskan apakah KB = α untuk beberapa kalimat α.

7. Truth Table Enumeration

Ini adalah pendekatan pengecekan model yang seperti namanya yaitu, menghitung semua model yang mungkin di mana KB bernilai benar dan memeriksa apakah α juga bernilai benar dalam model-model ini. Kami mencantumkan n simbol-simbol dalam KB dan menghitung model kedua tersebut secara mendalam dan memeriksa kebenaran KB dan α.

.

In [22]:

```
psource(tt_check_all)
```

```
def tt_check_all(kb, alpha, symbols, model):
    """Auxiliary routine to implement tt_entails."""
    if not symbols:
        if pl_true(kb, model):
            result = pl_true(alpha, model)
            assert result in (True, False)
            return result
        else:
            return True
    else:
        P, rest = symbols[0], symbols[1:]
        return (tt_check_all(kb, alpha, rest, extend(model, P, True)) and
                tt_check_all(kb, alpha, rest, extend(model, P, False)))
```

Singkatnya, tt_check_all mengevaluasi ekspresi logis ini untuk setiap model

pl_true(kb, model) => pl_true(alpha, model)

yang secara logis setara dengan

pl_true(kb, model) & ~pl_true(alpha, model)

yaitu, basis pengetahuan dan negasi kueri secara logis tidak konsisten.

tt_entails() hanya mengekstrak simbol dari kueri dan memanggil tt_check_all() dengan parameter yang tepat.

In [23]:

```
psource(tt_entails)
```

```
def tt_entails(kb, alpha):
    """Does kb entail the sentence alpha? Use truth tables. For propositional
    kb's and sentences. [Figure 7.10]. Note that the 'kb' should be an
    Expr which is a conjunction of clauses.
    >>> tt_entails(expr('P & Q'), expr('Q'))
    True
    """
    assert not variables(alpha)
    symbols = list(prop_symbols(kb & alpha))
    return tt_check_all(kb, alpha, symbols, {})
```

Perlu diingat bahwa untuk dua simbol P dan Q, P => Q bernilai salah hanya jika P bernilai Benar dan Q bernilai Salah. Contoh penggunaan tt_entails()

p	q	p→q
T	T	T
T	F	F
F	T	T
F	F	T

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Calcworkshop.com

P & Q bernilai Benar hanya jika P dan Q bernilai Benar. Oleh karena itu, (P & Q) => Q bernilai Benar

In [25]:

```
tt_entails(P | Q, Q)
```

Out[25]:

```
False
```

In [26]:

```
tt_entails(P | Q, P)
```

Out[26]:

```
False
```

Jika kita tahu bahwa P | Q benar, kita tidak dapat menyimpulkan nilai kebenaran P dan Q. Oleh karena itu (P | Q) => Q adalah Salah dan begitu juga (P | Q) => P.

In [27]:

```
(A, B, C, D, E, F, G) = symbols('A, B, C, D, E, F, G')
tt_entails(A & (B | C) & D & E & ~(F | G), A & D & E & ~F & ~G)
```

Out[27]:

```
True
```

Kita dapat melihat bahwa agar KB benar, A, D, E harus Benar dan F dan G harus Salah. Tidak ada yang dapat dikatakan tentang B atau C.

Berikut merupakan Pseudocode dari Code tt_entails:

FUNGSI tt_check_all(kb, alpha, symbols, model)

JIKA symbols kosong MAKA

JIKA pl_true(kb, model) MAKA

KEMBALI pl_true(alpha, model)

LAINNYA

KEMBALI True DAN False (yang bernilai False)

SELESAI JIKA

LAINNYA

P <- PERTAMA(symbols)

sisanya <- SISA(symbols)

KEMBALI tt_check_all(kb, alpha, sisanya, extend(model, P, True)) DAN

tt_check_all(kb, alpha, sisanya, extend(model, P, False))

SELESAI JIKA

END FUNGSI

Periksa apa yang wumpus_kb katakan Tentang P11

In [28]:

```
wumpus_kb.ask_if_true(~P11), wumpus_kb.ask_if_true(P11)
```

Out[28]:

```
(True, False)
```

Kita bisa lihat bahwa jika Knowledge base bernilai True, maka P11 bernilai False. Hal tersebut dikarenakan fungsi ask_if_true() mengembalikan True untuk α = ~P11 dan False unutk α = P11

8. Proof by Resolution

Teknik ini menggunakan pembuktian dengan kontradiksi, yaitu asumsi yang salah dibuktikan berlawanan dengan aksioma yang ada. Kontradiksi diperoleh melalui kesimpulan valid menggunakan aturan inferensi, khususnya aturan resolusi.

Ada satu kendala, algoritma yang menerapkan pembuktian dengan resolusi tidak dapat menangani kalimat kompleks. Implikasi dan bi-implikasi harus disederhanakan menjadi klausa yang lebih sederhana. Kita sudah tahu bahwa setiap kalimat dari logika proposisional secara logis setara dengan konjungsi klausa. Kita akan menggunakan fakta ini untuk keuntungan kita dan menyederhanakan kalimat input ke dalam bentuk Conjunctive Normal Form (CNF) yang merupakan konjungsi disjungsi literal. Misalnya:

$(P \rightarrow Q) \land (\neg R \lor S)$

Langkah-langkah menyederhanakan kalimat ini menjadi CNF:

1. Hilangkan implikasi (→):
    - P → Q setara dengan ¬P ∨ Q.
    - Jadi, kalimat menjadi: (¬P ∨ Q) ∧ (¬R ∨ S)
2. Kalimat sudah dalam bentuk CNF:

- Bentuk ini merupakan konjungsi (∧) dari disjungsi (∨) literal (¬P, Q, ¬R, S).

Kalimat akhir dalam CNF adalah:

(¬P ∨ Q) ∧ (¬R ∨ S)

Berikut Tabel Logical Equivalences

TABLE 6 Logical Equivalences.

Equivalence	*Name*
$p \wedge \mathbf{T} \equiv p$ $p \vee \mathbf{F} \equiv p$	Identity laws
$p \vee \mathbf{T} \equiv \mathbf{T}$ $p \wedge \mathbf{F} \equiv \mathbf{F}$	Domination laws
$p \vee p \equiv p$ $p \wedge p \equiv p$	Idempotent laws
$\neg(\neg p) \equiv p$	Double negation law
$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$	Commutative laws
$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	Associative laws
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	Distributive laws
$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$	De Morgan's laws
$p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$	Absorption laws
$p \vee \neg p = \mathbf{T}$ $p \wedge \neg p \equiv \mathbf{F}$	Negation laws

Kita akan mengeksekusi konversi ini menggunakan fungsi to_cnf

Berikut Tampilan fungsinya

```
In [30]:
psource(to_cnf)
```

```
def to_cnf(s):
    """Convert a propositional logical sentence to conjunctive normal form.
    That is, to the form ((A | ~B | ...) & (B | C | ...) & ...) [p. 253]
    >>> to_cnf('~(B | C)')
    (~B & ~C)
    """
    s = expr(s)
    if isinstance(s, str):
        s = expr(s)
    s = eliminate_implications(s)  # Steps 1, 2 from p. 253
    s = move_not_inwards(s)  # Step 3
    return distribute_and_over_or(s)  # Step 4
```

- to_cnf :memanggil tiga subrutin.
- eliminate_implications: mengonversi bi-implications dan implication ke logical equivalents-nya.
- move_not_inwards: menghapus negasi dari pernyataan majemuk dan memindahkannya ke dalam menggunakan hukum De Morgan.
- distribute_and_over_or mendistribusikan disjungsi ke konjungsi.

Berikut tampilan fungsi dari penjelasan di atas:

- eliminate_implications(s)

```
def eliminate_implications(s):
    """Change implications into equivalent form with only &, |, and ~ as logical ope
rators."""
    s = expr(s)
    if not s.args or is_symbol(s.op):
        return s  # Atoms are unchanged.
    args = list(map(eliminate_implications, s.args))
    a, b = args[0], args[-1]
    if s.op == '==>':
        return b | ~a
    elif s.op == '<==':
        return a | ~b
    elif s.op == '<=>':
        return (a | ~b) & (b | ~a)
    elif s.op == '^':
        assert len(args) == 2  # TODO: relax this restriction
        return (a & ~b) | (~a & b)
    else:
        assert s.op in ('&', '|', '~')
        return Expr(s.op, *args)
```

- Move_not_inwards(s):

```
def move_not_inwards(s):
    """Rewrite sentence s by moving negation sign inward.
    >>> move_not_inwards(~(A | B))
    (~A & ~B)"""
    s = expr(s)
    if s.op == '~':
        def NOT(b):
            return move_not_inwards(~b)
        a = s.args[0]
        if a.op == '~':
            return move_not_inwards(a.args[0])  # ~~A ==> A
        if a.op == '&':
            return associate('|', list(map(NOT, a.args)))
        if a.op == '|':
            return associate('&', list(map(NOT, a.args)))
        return s
    elif is_symbol(s.op) or not s.args:
        return s
    else:
        return Expr(s.op, *list(map(move_not_inwards, s.args)))
```

- distribute_and_over_or(s):

```
def distribute_and_over_or(s):
    """Given a sentence s consisting of conjunctions and disjunctions
    of literals, return an equivalent sentence in CNF.
    >>> distribute_and_over_or((A & B) | C)
    ((A | C) & (B | C))
    """
    s = expr(s)
    if s.op == '|':
        s = associate('|', s.args)
        if s.op != '|':
            return distribute_and_over_or(s)
        if len(s.args) == 0:
            return False
        if len(s.args) == 1:
            return distribute_and_over_or(s.args[0])
        conj = first(arg for arg in s.args if arg.op == '&')
        if not conj:
            return s
        others = [a for a in s.args if a is not conj]
        rest = associate('|', others)
        return associate('&', [distribute_and_over_or(c | rest)
                               for c in conj.args])
    elif s.op == '&':
        return associate('&', list(map(distribute_and_over_or, s.args)))
    else:
        return s
```

Mari lihat bagaimana Fungsi di atas berkerja:

In [32]:
```
A, B, C, D = expr('A, B, C, D')
to_cnf(A |'<=>'| B)
```

Out[32]:
```
((A | ~B) & (B | ~A))
```

In [33]:
```
to_cnf(A |'<=>'| (B & C))
```

Out[33]:
```
((A | ~B | ~C) & (B | ~A) & (C | ~A))
```

In [34]:
```
to_cnf(A & (B | (C & D)))
```

Out[34]:
```
(A & (C | B) & (D | B))
```

In [35]:
```
to_cnf((A |'<=>'| ~B) |'==>'| (C | ~D))
```

Out[35]:
```
((B | ~A | C | ~D) & (A | ~A | C | ~D) & (B | ~B | C | ~D) & (A | ~B | C | ~D))
```

Kembali ke masalah resolusi kita, kita dapat melihat bagaimana fungsi to_cnf digunakan di sini

In [36]:
```
psource(pl_resolution)
```

```
def pl_resolution(KB, alpha):
    """Propositional-logic resolution: say if alpha follows from KB. [Figure 7.1
2]"""
    clauses = KB.clauses + conjuncts(to_cnf(~alpha))
    new = set()
    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j])
                 for i in range(n) for j in range(i+1, n)]
        for (ci, cj) in pairs:
            resolvents = pl_resolve(ci, cj)
            if False in resolvents:
                return True
            new = new.union(set(resolvents))
        if new.issubset(set(clauses)):
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)
```

In [37]:
```
pl_resolution(wumpus_kb, ~P11), pl_resolution(wumpus_kb, P11)
```

Out[37]:
```
(True, False)
```

In [38]:
```
pl_resolution(wumpus_kb, ~P22), pl_resolution(wumpus_kb, P22)
```

Out[38]:
```
(False, False)
```

9. Forward and Bacward chaining

Sebelumnya, Kita telah menjelaskan dua algoritma untuk memeriksa apakah sebuah kalimat dapat diturunkan dari basis pengetahuan (KB). Tujuan utama dari algoritma ketiga ini adalah untuk menentukan apakah basis pengetahuan yang terdiri dari Horn Clauses memerlukan simbol kalimat tunggal q sebagai query. Namun, algoritma ini memiliki keterbatasan penting. Basis pengetahuan hanya boleh berisi frasa Horn, yaitu frasa yang mengandung paling banyak satu literal positif.

Horn Clauses

Horn clause adalah rumus logika yang memiliki sifat-sifat tertentu sehingga dapat digunakan dalam berbagai bidang, seperti pemrograman logika, spesifikasi formal, aljabar universal, dan teori model.

Type of Horn clause	Disjunction form	Implication form	Read intuitively as
Definite clause	$\neg p \vee \neg q \vee \ldots \vee \neg t \vee u$	$u \leftarrow p \wedge q \wedge \ldots \wedge t$	assume that, if p and q and ... and t all hold, then also u holds
Fact	u	$u \leftarrow true$	assume that u holds
Goal clause	$\neg p \vee \neg q \vee \ldots \vee \neg t$	$false \leftarrow p \wedge q \wedge \ldots \wedge t$	show that p and q and ... and t all hold[5]

Tabel di atas didapatkan dari Wikipedia. Dari tabel di atas dapat kita ketahui beberapa ciri ciri dari Horn Clauses

Horn clause memiliki beberapa ciri, yaitu:

- Horn clause adalah disjungsi literal yang paling banyak memiliki satu literal yang tidak dinegasikan.
- Horn clause yang memiliki tepat satu literal yang tidak dinegasikan disebut definite clause.
- Horn clause yang tidak memiliki literal positif disebut goal clause.
- Horn clause yang tidak memiliki literal negatif disebut logic fact.
- Horn clause merupakan fondasi logika dari bahasa pemrograman Prolog

Fungsi **pl_fc_entails** menerapkan forward chaining untuk memeriksa apakah basis pengetahuan (KB) menyiratkan simbol q. Sebelum melanjutkan, perlu dicatat bahwa **pl_fc_entails** tidak menggunakan instance KB biasa. Basis pengetahuan di sini adalah instance dari kelas **PropDefiniteKB**, yang diturunkan dari kelas PropKB, tetapi dimodifikasi untuk menyimpan klausa pasti (definite clauses). Perbedaan utamanya adalah adanya metode pembantu di **PropDefiniteKB** yang mengembalikan daftar klausa dalam KB yang memiliki simbol p di premisnya.

In [39]:
```
psource(PropDefiniteKB.clauses_with_premise)
```

```
def clauses_with_premise(self, p):
    """Return a list of the clauses in KB that have p in their premise.
    This could be cached away for O(1) speed, but we'll recompute it."""
    return [c for c in self.clauses
            if c.op == '==>' and p in conjuncts(c.args[0])]
```

In [40]:
```
psource(pl_fc_entails)
```

```
def pl_fc_entails(KB, q):
    """Use forward chaining to see if a PropDefiniteKB entails symbol q.
    [Figure 7.15]
    >>> pl_fc_entails(horn_clauses_KB, expr('Q'))
    True
    """
    count = {c: len(conjuncts(c.args[0]))
             for c in KB.clauses
             if c.op == '==>'}
    inferred = defaultdict(bool)
    agenda = [s for s in KB.clauses if is_prop_symbol(s.op)]
    while agenda:
        p = agenda.pop()
        if p == q:
            return True
        if not inferred[p]:
            inferred[p] = True
            for c in KB.clauses_with_premise(p):
                count[c] -= 1
                if count[c] == 0:
                    agenda.append(c.args[1])
    return False
```

Penjelasan kode:

Fungsi ini menerima basis pengetahuan (KB) berupa instance dari **PropDefiniteKB** dan sebuah kueri **q** sebagai input.

- **Count** awalnya menyimpan jumlah simbol dalam premis setiap kalimat dalam basis pengetahuan.
- **Conjuncts** memisahkan kalimat yang diberikan pada konjungsi.
- **Inferred** diinisialisasi sebagai boolean defaultdict. Ini akan digunakan nanti untuk memeriksa apakah kita telah menyimpulkan semua premis dari setiap klausa agenda.
- **agenda** awalnya menyimpan daftar klausa yang diketahui basis pengetahuan sebagai benar.
- **is_prop_symbol** memeriksa apakah simbol yang diberikan adalah simbol logika proposisional yang valid.

Fungsi ini menerima basis pengetahuan (KB) dari **PropDefiniteKB** dan kueri **q**. Variabel **count** menyimpan jumlah simbol dalam premis tiap kalimat, sementara **inferred** digunakan untuk memeriksa apakah semua premis telah disimpulkan. **agenda** berisi klausa yang diketahui benar, dan setiap simbol **p** diproses. Jika **p** sama dengan **q**, penyimpulan berlaku. Saat **count** mencapai nol, kesimpulan ditambahkan ke agenda. Metode **clauses_with_premise** dari **PropKB** mengembalikan daftar klausa dengan premis **p**.

Setelah memahami cara kerja fungsi ini, mari kita lihat beberapa contoh penggunaannya, tetapi pertama-tama kita perlu mendefinisikan basis pengetahuan kita. Kita berasumsi bahwa klausa-klausa berikut diketahui benar.

In [41]:
```
clauses = ['(B & F)==>E',
           '(A & E & F)==>G',
           '(B & C)==>F',
           '(A & B)==>D',
           '(E & F)==>H',
           '(H & I)==>J',
           'A',
           'B',
           'C']
```

Kita akan memberi tahu informasi berikut kepada Knowledge Base

In [42]:
```
definite_clauses_KB = PropDefiniteKB()
for clause in clauses:
    definite_clauses_KB.tell(expr(clause))
```

We can now check if our knowledge base entails the following queries.

In [43]:
```
pl_fc_entails(definite_clauses_KB, expr('G'))
```

Out[43]: True

In [44]:
```
pl_fc_entails(definite_clauses_KB, expr('H'))
```

Out[44]: True

In [45]:
```
pl_fc_entails(definite_clauses_KB, expr('I'))
```

Out[45]: False

In [46]:
```
pl_fc_entails(definite_clauses_KB, expr('J'))
```

Out[46]: False