Logic

Notebook Jupyter ini berfungsi sebagai materi pendukung untuk topik-topik yang dibahas dalam **Bab 6 Logical Agents**, **Bab 7 First-Order Logic** and **Bab 8 Inference in First-Order Logic** dari buku *Artificial Intelligence: A Modern Approach*. Kami menggunakan implementasi dalam modul logic.py. Lihat panduan praktikum logic untuk petunjuk lebih lanjut.

Mari kita impor terlebih dahulu semua yang ada dalam modul logika.

In [100…

```python
%pip install ipythonblocks
%pip install qpsolvers
from utils import *
from logic import *
from notebook import psource
```

```
Requirement already satisfied: ipythonblocks in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (1.9.1)
Requirement already satisfied: ipython>=4.0 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from ipythonblocks) (9.5.0)
Requirement already satisfied: notebook>=4.0 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from ipythonblocks) (7.4.5)
Requirement already satisfied: requests>=1.0 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from ipythonblocks) (2.32.5)
Requirement already satisfied: colorama in d:\ai_journey\computational_intelligen
ce\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (0.4.6)
Requirement already satisfied: decorator in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (5.2.1)
Requirement already satisfied: ipython-pygments-lexers in d:\ai_journey\computati
onal_intelligence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (1.1.1)
Requirement already satisfied: jedi>=0.16 in d:\ai_journey\computational_intellig
ence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (0.19.2)
Requirement already satisfied: matplotlib-inline in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (0.1.7)
Requirement already satisfied: prompt_toolkit<3.1.0,>=3.0.41 in d:\ai_journey\com
putational_intelligence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks)
(3.0.52)
Requirement already satisfied: pygments>=2.4.0 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (2.19.2)
Requirement already satisfied: stack_data in d:\ai_journey\computational_intellig
ence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (0.6.3)
Requirement already satisfied: traitlets>=5.13.0 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from ipython>=4.0->ipythonblocks) (5.14.3)
Requirement already satisfied: wcwidth in d:\ai_journey\computational_intelligenc
e\ci\lib\site-packages (from prompt_toolkit<3.1.0,>=3.0.41->ipython>=4.0->ipython
blocks) (0.2.13)
Requirement already satisfied: parso<0.9.0,>=0.8.4 in d:\ai_journey\computational
_intelligence\ci\lib\site-packages (from jedi>=0.16->ipython>=4.0->ipythonblocks)
(0.8.5)
Requirement already satisfied: jupyter-server<3,>=2.4.0 in d:\ai_journey\computat
ional_intelligence\ci\lib\site-packages (from notebook>=4.0->ipythonblocks) (2.1
7.0)
Requirement already satisfied: jupyterlab-server<3,>=2.27.1 in d:\ai_journey\comp
utational_intelligence\ci\lib\site-packages (from notebook>=4.0->ipythonblocks)
(2.27.3)
Requirement already satisfied: jupyterlab<4.5,>=4.4.5 in d:\ai_journey\computatio
nal_intelligence\ci\lib\site-packages (from notebook>=4.0->ipythonblocks) (4.4.7)
Requirement already satisfied: notebook-shim<0.3,>=0.2 in d:\ai_journey\computati
onal_intelligence\ci\lib\site-packages (from notebook>=4.0->ipythonblocks) (0.2.
4)
Requirement already satisfied: tornado>=6.2.0 in d:\ai_journey\computational_inte
lligence\ci\lib\site-packages (from notebook>=4.0->ipythonblocks) (6.5.2)
Requirement already satisfied: anyio>=3.1.0 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipytho
nblocks) (4.10.0)
Requirement already satisfied: argon2-cffi>=21.1 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->i
pythonblocks) (25.1.0)
Requirement already satisfied: jinja2>=3.0.3 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipyth
onblocks) (3.1.6)
Requirement already satisfied: jupyter-client>=7.4.4 in d:\ai_journey\computation
al_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.
0->ipythonblocks) (8.6.3)
Requirement already satisfied: jupyter-core!=5.0.*,>=4.12 in d:\ai_journey\comput
ational_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->noteboo
```

```
k>=4.0->ipythonblocks) (5.8.1)
Requirement already satisfied: jupyter-events>=0.11.0 in d:\ai_journey\computatio
nal_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=
4.0->ipythonblocks) (0.12.0)
Requirement already satisfied: jupyter-server-terminals>=0.4.4 in d:\ai_journey\c
omputational_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->no
tebook>=4.0->ipythonblocks) (0.5.3)
Requirement already satisfied: nbconvert>=6.4.4 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ip
ythonblocks) (7.16.6)
Requirement already satisfied: nbformat>=5.3.0 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipy
thonblocks) (5.10.4)
Requirement already satisfied: packaging>=22.0 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipy
thonblocks) (25.0)
Requirement already satisfied: prometheus-client>=0.9 in d:\ai_journey\computatio
nal_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=
4.0->ipythonblocks) (0.22.1)
Requirement already satisfied: pywinpty>=2.0.1 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipy
thonblocks) (3.0.0)
Requirement already satisfied: pyzmq>=24 in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonbl
ocks) (27.0.2)
Requirement already satisfied: send2trash>=1.8.2 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->i
pythonblocks) (1.8.3)
Requirement already satisfied: terminado>=0.8.3 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.0->ip
ythonblocks) (0.18.1)
Requirement already satisfied: websocket-client>=1.7 in d:\ai_journey\computation
al_intelligence\ci\lib\site-packages (from jupyter-server<3,>=2.4.0->notebook>=4.
0->ipythonblocks) (1.8.0)
Requirement already satisfied: async-lru>=1.0.0 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from jupyterlab<4.5,>=4.4.5->notebook>=4.0->ipyt
honblocks) (2.0.5)
Requirement already satisfied: httpx<1,>=0.25.0 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from jupyterlab<4.5,>=4.4.5->notebook>=4.0->ipyt
honblocks) (0.28.1)
Requirement already satisfied: ipykernel!=6.30.0,>=6.5.0 in d:\ai_journey\computa
tional_intelligence\ci\lib\site-packages (from jupyterlab<4.5,>=4.4.5->notebook>=
4.0->ipythonblocks) (6.30.1)
Requirement already satisfied: jupyter-lsp>=2.0.0 in d:\ai_journey\computational_
intelligence\ci\lib\site-packages (from jupyterlab<4.5,>=4.4.5->notebook>=4.0->ip
ythonblocks) (2.3.0)
Requirement already satisfied: setuptools>=41.1.0 in d:\ai_journey\computational_
intelligence\ci\lib\site-packages (from jupyterlab<4.5,>=4.4.5->notebook>=4.0->ip
ythonblocks) (80.9.0)
Requirement already satisfied: certifi in d:\ai_journey\computational_intelligenc
e\ci\lib\site-packages (from httpx<1,>=0.25.0->jupyterlab<4.5,>=4.4.5->notebook>=
4.0->ipythonblocks) (2025.8.3)
Requirement already satisfied: httpcore==1.* in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from httpx<1,>=0.25.0->jupyterlab<4.5,>=4.4.5->note
book>=4.0->ipythonblocks) (1.0.9)
Requirement already satisfied: idna in d:\ai_journey\computational_intelligence\c
i\lib\site-packages (from httpx<1,>=0.25.0->jupyterlab<4.5,>=4.4.5->notebook>=4.0
->ipythonblocks) (3.10)
Requirement already satisfied: h11>=0.16 in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (from httpcore==1.*->httpx<1,>=0.25.0->jupyterlab<4.5,>=
```

```
4.4.5->notebook>=4.0->ipythonblocks) (0.16.0)
Requirement already satisfied: babel>=2.10 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from jupyterlab-server<3,>=2.27.1->notebook>=4.0->ipy
thonblocks) (2.17.0)
Requirement already satisfied: json5>=0.9.0 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from jupyterlab-server<3,>=2.27.1->notebook>=4.0->ip
ythonblocks) (0.12.1)
Requirement already satisfied: jsonschema>=4.18.0 in d:\ai_journey\computational_
intelligence\ci\lib\site-packages (from jupyterlab-server<3,>=2.27.1->notebook>=
4.0->ipythonblocks) (4.25.1)
Requirement already satisfied: sniffio>=1.1 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from anyio>=3.1.0->jupyter-server<3,>=2.4.0->noteboo
k>=4.0->ipythonblocks) (1.3.1)
Requirement already satisfied: typing_extensions>=4.5 in d:\ai_journey\computatio
nal_intelligence\ci\lib\site-packages (from anyio>=3.1.0->jupyter-server<3,>=2.4.
0->notebook>=4.0->ipythonblocks) (4.15.0)
Requirement already satisfied: argon2-cffi-bindings in d:\ai_journey\computationa
l_intelligence\ci\lib\site-packages (from argon2-cffi>=21.1->jupyter-server<3,>=
2.4.0->notebook>=4.0->ipythonblocks) (25.1.0)
Requirement already satisfied: comm>=0.1.1 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from ipykernel!=6.30.0,>=6.5.0->jupyterlab<4.5,>=4.4.
5->notebook>=4.0->ipythonblocks) (0.2.3)
Requirement already satisfied: debugpy>=1.6.5 in d:\ai_journey\computational_inte
lligence\ci\lib\site-packages (from ipykernel!=6.30.0,>=6.5.0->jupyterlab<4.5,>=
4.4.5->notebook>=4.0->ipythonblocks) (1.8.16)
Requirement already satisfied: nest-asyncio>=1.4 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from ipykernel!=6.30.0,>=6.5.0->jupyterlab<4.5,
>=4.4.5->notebook>=4.0->ipythonblocks) (1.6.0)
Requirement already satisfied: psutil>=5.7 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from ipykernel!=6.30.0,>=6.5.0->jupyterlab<4.5,>=4.4.
5->notebook>=4.0->ipythonblocks) (7.0.0)
Requirement already satisfied: MarkupSafe>=2.0 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from jinja2>=3.0.3->jupyter-server<3,>=2.4.0->not
ebook>=4.0->ipythonblocks) (3.0.2)
Requirement already satisfied: attrs>=22.2.0 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from jsonschema>=4.18.0->jupyterlab-server<3,>=2.2
7.1->notebook>=4.0->ipythonblocks) (25.3.0)
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in d:\ai_jour
ney\computational_intelligence\ci\lib\site-packages (from jsonschema>=4.18.0->jup
yterlab-server<3,>=2.27.1->notebook>=4.0->ipythonblocks) (2025.4.1)
Requirement already satisfied: referencing>=0.28.4 in d:\ai_journey\computational
_intelligence\ci\lib\site-packages (from jsonschema>=4.18.0->jupyterlab-server<3,
>=2.27.1->notebook>=4.0->ipythonblocks) (0.36.2)
Requirement already satisfied: rpds-py>=0.7.1 in d:\ai_journey\computational_inte
lligence\ci\lib\site-packages (from jsonschema>=4.18.0->jupyterlab-server<3,>=2.2
7.1->notebook>=4.0->ipythonblocks) (0.27.1)
Requirement already satisfied: python-dateutil>=2.8.2 in d:\ai_journey\computatio
nal_intelligence\ci\lib\site-packages (from jupyter-client>=7.4.4->jupyter-server
<3,>=2.4.0->notebook>=4.0->ipythonblocks) (2.9.0.post0)
Requirement already satisfied: platformdirs>=2.5 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from jupyter-core!=5.0.*,>=4.12->jupyter-server
<3,>=2.4.0->notebook>=4.0->ipythonblocks) (4.4.0)
Requirement already satisfied: pywin32>=300 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from jupyter-core!=5.0.*,>=4.12->jupyter-server<3,>=
2.4.0->notebook>=4.0->ipythonblocks) (311)
Requirement already satisfied: python-json-logger>=2.0.4 in d:\ai_journey\computa
tional_intelligence\ci\lib\site-packages (from jupyter-events>=0.11.0->jupyter-se
rver<3,>=2.4.0->notebook>=4.0->ipythonblocks) (3.3.0)
Requirement already satisfied: pyyaml>=5.3 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from jupyter-events>=0.11.0->jupyter-server<3,>=2.4.0
```

```
->notebook>=4.0->ipythonblocks) (6.0.2)
Requirement already satisfied: rfc3339-validator in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from jupyter-events>=0.11.0->jupyter-server<3,>
=2.4.0->notebook>=4.0->ipythonblocks) (0.1.4)
Requirement already satisfied: rfc3986-validator>=0.1.1 in d:\ai_journey\computat
ional_intelligence\ci\lib\site-packages (from jupyter-events>=0.11.0->jupyter-ser
ver<3,>=2.4.0->notebook>=4.0->ipythonblocks) (0.1.1)
Requirement already satisfied: fqdn in d:\ai_journey\computational_intelligence\c
i\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jupyter-events>=0.1
1.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.5.1)
Requirement already satisfied: isoduration in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jupyter-event
s>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (20.11.0)
Requirement already satisfied: jsonpointer>1.13 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jupyter-
events>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (3.0.0)
Requirement already satisfied: rfc3987-syntax>=1.1.0 in d:\ai_journey\computation
al_intelligence\ci\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jup
yter-events>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.
1.0)
Requirement already satisfied: uri-template in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jupyter-even
ts>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.3.0)
Requirement already satisfied: webcolors>=24.6.0 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from jsonschema[format-nongpl]>=4.18.0->jupyter
-events>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (24.11.
1)
Requirement already satisfied: beautifulsoup4 in d:\ai_journey\computational_inte
lligence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.4.0->n
otebook>=4.0->ipythonblocks) (4.13.5)
Requirement already satisfied: bleach!=5.0.0 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from bleach[css]!=5.0.0->nbconvert>=6.4.4->jupyter-
server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (6.2.0)
Requirement already satisfied: defusedxml in d:\ai_journey\computational_intellig
ence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.4.0->noteb
ook>=4.0->ipythonblocks) (0.7.1)
Requirement already satisfied: jupyterlab-pygments in d:\ai_journey\computational
_intelligence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.
4.0->notebook>=4.0->ipythonblocks) (0.3.0)
Requirement already satisfied: mistune<4,>=2.0.3 in d:\ai_journey\computational_i
ntelligence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.4.0
->notebook>=4.0->ipythonblocks) (3.1.4)
Requirement already satisfied: nbclient>=0.5.0 in d:\ai_journey\computational_int
elligence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.4.0->
notebook>=4.0->ipythonblocks) (0.10.2)
Requirement already satisfied: pandocfilters>=1.4.1 in d:\ai_journey\computationa
l_intelligence\ci\lib\site-packages (from nbconvert>=6.4.4->jupyter-server<3,>=2.
4.0->notebook>=4.0->ipythonblocks) (1.5.1)
Requirement already satisfied: webencodings in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from bleach!=5.0.0->bleach[css]!=5.0.0->nbconvert>=
6.4.4->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (0.5.1)
Requirement already satisfied: tinycss2<1.5,>=1.1.0 in d:\ai_journey\computationa
l_intelligence\ci\lib\site-packages (from bleach[css]!=5.0.0->nbconvert>=6.4.4->j
upyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.4.0)
Requirement already satisfied: fastjsonschema>=2.15 in d:\ai_journey\computationa
l_intelligence\ci\lib\site-packages (from nbformat>=5.3.0->jupyter-server<3,>=2.
4.0->notebook>=4.0->ipythonblocks) (2.21.2)
Requirement already satisfied: six>=1.5 in d:\ai_journey\computational_intelligen
ce\ci\lib\site-packages (from python-dateutil>=2.8.2->jupyter-client>=7.4.4->jupy
ter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.17.0)
```

```
Requirement already satisfied: charset_normalizer<4,>=2 in d:\ai_journey\computat
ional_intelligence\ci\lib\site-packages (from requests>=1.0->ipythonblocks) (3.4.
3)
Requirement already satisfied: urllib3<3,>=1.21.1 in d:\ai_journey\computational_
intelligence\ci\lib\site-packages (from requests>=1.0->ipythonblocks) (2.5.0)
Requirement already satisfied: lark>=1.2.2 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from rfc3987-syntax>=1.1.0->jsonschema[format-nongpl]
>=4.18.0->jupyter-events>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipytho
nblocks) (1.2.2)
Requirement already satisfied: cffi>=1.0.1 in d:\ai_journey\computational_intelli
gence\ci\lib\site-packages (from argon2-cffi-bindings->argon2-cffi>=21.1->jupyter
-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (1.17.1)
Requirement already satisfied: pycparser in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (from cffi>=1.0.1->argon2-cffi-bindings->argon2-cffi>=2
1.1->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks) (2.22)
Requirement already satisfied: soupsieve>1.2 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from beautifulsoup4->nbconvert>=6.4.4->jupyter-serv
er<3,>=2.4.0->notebook>=4.0->ipythonblocks) (2.8)
Requirement already satisfied: arrow>=0.15.0 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from isoduration->jsonschema[format-nongpl]>=4.18.0
->jupyter-events>=0.11.0->jupyter-server<3,>=2.4.0->notebook>=4.0->ipythonblocks)
(1.3.0)
Requirement already satisfied: types-python-dateutil>=2.8.10 in d:\ai_journey\com
putational_intelligence\ci\lib\site-packages (from arrow>=0.15.0->isoduration->js
onschema[format-nongpl]>=4.18.0->jupyter-events>=0.11.0->jupyter-server<3,>=2.4.0
->notebook>=4.0->ipythonblocks) (2.9.0.20250822)
Requirement already satisfied: executing>=1.2.0 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from stack_data->ipython>=4.0->ipythonblocks)
(2.2.1)
Requirement already satisfied: asttokens>=2.1.0 in d:\ai_journey\computational_in
telligence\ci\lib\site-packages (from stack_data->ipython>=4.0->ipythonblocks)
(3.0.0)
Requirement already satisfied: pure-eval in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (from stack_data->ipython>=4.0->ipythonblocks) (0.2.3)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: qpsolvers in d:\ai_journey\computational_intellige
nce\ci\lib\site-packages (4.8.1)
Requirement already satisfied: numpy>=1.15.4 in d:\ai_journey\computational_intel
ligence\ci\lib\site-packages (from qpsolvers) (2.3.2)
Requirement already satisfied: scipy>=1.2.0 in d:\ai_journey\computational_intell
igence\ci\lib\site-packages (from qpsolvers) (1.16.1)
Note: you may need to restart the kernel to use updated packages.
```

CONTENTS

- Logical sentences
    - Expr
    - PropKB
    - Knowledge-based agents
    - Inference in propositional knowledge base
        - Truth table enumeration
        - Proof by resolution
        - Forward and backward chaining
        - DPLL
        - WalkSAT

- SATPlan
  - FolKB
  - Inference in first order knowledge base
    - Unification
    - Forward chaining algorithm
    - Backward chaining algorithm

Logical Sentences

The `Expr` class is designed to represent any kind of mathematical expression. The simplest type of `Expr` is a symbol, which can be defined with the function `Symbol`:

```
In [101…   Symbol('x')
```

```
Out[101…   x
```

Or we can define multiple symbols at the same time with the function `symbols`:

```
In [102…   (x, y, P, Q, f) = symbols('x, y, P, Q, f')
```

We can combine `Expr`s with the regular Python infix and prefix operators. Here's how we would form the logical sentence "P and not Q":

```
In [103…   P & ~Q
```

```
Out[103…   (P & ~Q)
```

This works because the `Expr` class overloads the `&` operator with this definition:

```
def __and__(self, other): return Expr('&',  self, other)```
```

and does similar overloads `for` the other operators. An `Expr` has two fields: `op` `for` the operator, which `is` always a string, `and` `args` `for` the arguments, which `is` a tuple of 0 or more expressions. By "expression," I mean either an instance of `Expr`, or a number. Let's take a look at the fields for some `Expr` examples:

```
In [104…   sentence = P & ~Q

           sentence.op
```

```
Out[104…   '&'
```

```
In [105…   sentence.args
```

```
Out[105…   (P, ~Q)
```

```
In [106…   P.op
```

```
Out[106…   'P'
```

```
In [107…   P.args
```

```
Out[107…   ()
```

```
In [108…   Pxy = P(x, y)

           Pxy.op
```

```
Out[108…   'P'
```

```
In [109…   Pxy.args
```

```
Out[109…   (x, y)
```

It is important to note that the `Expr` class does not define the *logic* of Propositional Logic sentences; it just gives you a way to *represent* expressions. Think of an `Expr` as an [abstract syntax tree](). Each of the `args` in an `Expr` can be either a symbol, a number, or a nested `Expr`. We can nest these trees to any depth. Here is a deply nested `Expr`:

```
In [110…   3 * f(x, y) + P(y) / 2 + 1
```

```
Out[110…   (((3 * f(x, y)) + (P(y) / 2)) + 1)
```

Operators for Constructing Logical Sentences

Here is a table of the operators that can be used to form sentences. Note that we have a problem: we want to use Python operators to make sentences, so that our programs (and our interactive sessions like the one here) will show simple code. But Python does not allow implication arrows as operators, so for now we have to use a more verbose notation that Python does allow: `|'==>'|` instead of just `==>`. Alternately, you can always use the more verbose `Expr` constructor forms:

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)`

Here's an example of defining a sentence with an implication arrow:

```
In [111…   ~(P & Q)  |'==>'|  (~P | ~Q)
```

```
Out[111…   (~(P & Q) ==> (~P | ~Q))
```

`expr`: a Shortcut for Constructing Sentences

If the `|'==>'|` notation looks ugly to you, you can use the function `expr` instead:

```
In [112...  expr('~(P & Q)  ==>  (~P | ~Q)')
```

```
Out[112...  (~(P & Q) ==> (~P | ~Q))
```

`expr` takes a string as input, and parses it into an `Expr`. The string can contain arrow operators: `==>`, `<==`, or `<=>`, which are handled as if they were regular Python infix operators. And `expr` automatically defines any symbols, so you don't need to pre-define them:

```
In [113...  expr('sqrt(b ** 2 - 4 * a * c)')
```

```
Out[113...  sqrt(((b ** 2) - ((4 * a) * c)))
```

For now that's all you need to know about `expr`. If you are interested, we explain the messy details of how `expr` is implemented and how `|'==>'|` is handled in the appendix.

Propositional Knowledge Bases: `PropKB`

The class `PropKB` can be used to represent a knowledge base of propositional logic sentences.

We see that the class `KB` has four methods, apart from `__init__`. A point to note here: the `ask` method simply calls the `ask_generator` method. Thus, this one has already been implemented, and what you'll have to actually implement when you create your own knowledge base class (though you'll probably never need to, considering the ones we've created for you) will be the `ask_generator` function and not the `ask` function itself.

The class `PropKB` now.

- `__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.

Wumpus World KB

Let us create a `PropKB` for the wumpus world with the sentences mentioned in `section 7.4.3`.

```
In [114… 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 [115… P11, P12, P21, P22, P31, B11, B21 = expr('P11, P12, P21, P22, P31, B11, B21')
```

Now we tell sentences based on `section 7.4.3`.
There is no pit in `[1,1]`.

```
In [116… wumpus_kb.tell(~P11)
```

A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square but for now, we include just the relevant squares.

```
In [117… wumpus_kb.tell(B11 | '<=>' | ((P12 | P21)))
         wumpus_kb.tell(B21 | '<=>' | ((P11 | P22 | P31)))
```

Now we include the breeze percepts for the first two squares leading up to the situation in `Figure 7.3(b)`

```
In [118… wumpus_kb.tell(~B11)
         wumpus_kb.tell(B21)
```

We can check the clauses stored in a `KB` by accessing its `clauses` variable

```
In [119… wumpus_kb.clauses
```

```
Out[119…    [~P11,
             (~P12 | B11),
             (~P21 | B11),
             (P12 | P21 | ~B11),
             (~P11 | B21),
             (~P22 | B21),
             (~P31 | B21),
             (P11 | P22 | P31 | ~B21),
             ~B11,
             B21]
```

We see that the equivalence $B_{1,1} \iff (P_{1,2} \lor P_{2,1})$ was automatically converted to two implications which were inturn converted to CNF which is stored in the `KB`.

$B_{1,1} \iff (P_{1,2} \lor P_{2,1})$ was split into $B_{1,1} \implies (P_{1,2} \lor P_{2,1})$ and $B_{1,1} \impliedby (P_{1,2} \lor P_{2,1})$.

$B_{1,1} \implies (P_{1,2} \lor P_{2,1})$ was converted to $P_{1,2} \lor P_{2,1} \lor \neg B_{1,1}$.

$B_{1,1} \impliedby (P_{1,2} \lor P_{2,1})$ was converted to $\neg(P_{1,2} \lor P_{2,1}) \lor B_{1,1}$ which becomes $(\neg P_{1,2} \lor B_{1,1}) \land (\neg P_{2,1} \lor B_{1,1})$ after applying De Morgan's laws and distributing the disjunction.

$B_{2,1} \iff (P_{1,1} \lor P_{2,2} \lor P_{3,2})$ is converted in similar manner.

Knowledge based agents

A knowledge-based agent is a simple generic agent that maintains and handles a knowledge base. The knowledge base may initially contain some background knowledge. The purpose of a KB agent is to provide a level of abstraction over knowledge-base manipulation and is to be used as a base class for agents that work on a knowledge base. Given a percept, the KB agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

Our implementation of `KB-Agent` is encapsulated in a class `KB_AgentProgram` which inherits from the `KB` class.

Let's have a look.

```
In [120…   psource(KBAgentProgram)
```

```python
def KBAgentProgram(kb):
    """
    [Figure 7.1]
    A generic logical knowledge-based agent program.
    """
    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
```

The helper functions `make_percept_sentence` , `make_action_query` and `make_action_sentence` are all aptly named and as expected, `make_percept_sentence` makes first-order logic sentences about percepts we want our agent to receive, `make_action_query` asks the underlying `KB` about the action that should be taken and `make_action_sentence` tells the underlying `KB` about the action it has just taken.

Inference in Propositional Knowledge Base

In this section we will look at two algorithms to check if a sentence is entailed by the `KB` . Our goal is to decide whether $\mathrm{KB} \models \alpha$ for some sentence α.

Truth Table Enumeration

It is a model-checking approach which, as the name suggests, enumerates all possible models in which the `KB` is true and checks if α is also true in these models. We list the n symbols in the `KB` and enumerate the 2^n models in a depth-first manner and check the truth of `KB` and α.

```python
In [121… psource(tt_check_all)
```

```python
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)))
```

The algorithm basically computes every line of the truth table $KB \implies \alpha$ and checks if it is true everywhere.

If symbols are defined, the routine recursively constructs every combination of truth values for the symbols and then, it checks whether `model` is consistent with `kb`. The given models correspond to the lines in the truth table, which have a `true` in the KB column, and for these lines it checks whether the query evaluates to true `result = pl_true(alpha, model)`.

In short, `tt_check_all` evaluates this logical expression for each `model`
`pl_true(kb, model) => pl_true(alpha, model)`
which is logically equivalent to
`pl_true(kb, model) & ~pl_true(alpha, model)`
that is, the knowledge base and the negation of the query are logically inconsistent.

`tt_entails()` just extracts the symbols from the query and calls `tt_check_all()` with the proper parameters.

In [122... `psource(tt_entails)`

```python
def tt_entails(kb, alpha):
    """
    [Figure 7.10]
    Does kb entail the sentence alpha? Use truth tables. For propositional
    kb's and sentences. 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, {})
```

Keep in mind that for two symbols P and Q, P => Q is false only when P is `True` and Q is `False`. Example usage of `tt_entails()`:

In [123... `tt_entails(P & Q, Q)`

Out[123...   True

P & Q is True only when both P and Q are True. Hence, (P & Q) => Q is True

In [124... `tt_entails(P | Q, Q)`

Out[124...   False

In [125... `tt_entails(P | Q, P)`

Out[125...   False

If we know that P | Q is true, we cannot infer the truth values of P and Q. Hence (P | Q) => Q is False and so is (P | Q) => P.

In [126...
```python
(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[126...   True

We can see that for the KB to be true, A, D, E have to be True and F and G have to be False. Nothing can be said about B or C.

Coming back to our problem, note that `tt_entails()` takes an `Expr` which is a conjunction of clauses as the input instead of the `KB` itself. You can use the `ask_if_true()` method of `PropKB` which does all the required conversions. Let's check what `wumpus_kb` tells us about $P_{1,1}$.

In [127... `wumpus_kb.ask_if_true(~P11), wumpus_kb.ask_if_true(P11)`

Out[127...   (True, False)

Looking at Figure 7.9 we see that in all models in which the knowledge base is `True` , $P_{1,1}$ is `False` . It makes sense that `ask_if_true()` returns `True` for $\alpha = \neg P_{1,1}$ and `False` for $\alpha = P_{1,1}$. This begs the question, what if α is `True` in only a portion of all models. Do we return `True` or `False` ? This doesn't rule out the possibility of α being `True` but it is not entailed by the `KB` so we return `False` in such cases. We can see this is the case for $P_{2,2}$ and $P_{3,1}$.

In [128… ```
          wumpus_kb.ask_if_true(~P22), wumpus_kb.ask_if_true(P22)
          ```

Out[128…  `(False, False)`

## Proof by Resolution

Recall that our goal is to check whether $\text{KB} \models \alpha$ i.e. is $\text{KB} \implies \alpha$ true in every model. Suppose we wanted to check if $P \implies Q$ is valid. We check the satisfiability of $\neg(P \implies Q)$, which can be rewritten as $P \wedge \neg Q$. If $P \wedge \neg Q$ is unsatisfiable, then $P \implies Q$ must be true in all models. This gives us the result "$\text{KB} \models \alpha$ *if and only if* $\text{KB} \wedge \neg \alpha$ is unsatisfiable".

This technique corresponds to *proof by **contradiction***, a standard mathematical proof technique. We assume $\alpha$ to be false and show that this leads to a contradiction with known axioms in $\text{KB}$. We obtain a contradiction by making valid inferences using inference rules. In this proof we use a single inference rule, **resolution** which states $(l_1 \vee \cdots \vee l_k) \wedge (m_1 \vee \cdots \vee m_n) \wedge (l_i \iff \neg m_j) \implies l_1 \vee \cdots \vee l_{i-1} \vee l_{i+1} \vee \cdots \vee l_k$ . Applying the resolution yields us a clause which we add to the KB. We keep doing this until:

- There are no new clauses that can be added, in which case $\text{KB} \nvDash \alpha$.
- Two clauses resolve to yield the *empty clause*, in which case $\text{KB} \models \alpha$.

The *empty clause* is equivalent to *False* because it arises only from resolving two complementary unit clauses such as $P$ and $\neg P$ which is a contradiction as both $P$ and $\neg P$ can't be *True* at the same time.

There is one catch however, the algorithm that implements proof by resolution cannot handle complex sentences. Implications and bi-implications have to be simplified into simpler clauses. We already know that *every sentence of a propositional logic is logically equivalent to a conjunction of clauses*. We will use this fact to our advantage and simplify the input sentence into the **conjunctive normal form** (CNF) which is a conjunction of disjunctions of literals. For eg:

$$(A \vee B) \wedge (\neg B \vee C \vee \neg D) \wedge (D \vee \neg E)$$

This is equivalent to the POS (Product of sums) form in digital electronics. Here's an outline of how the conversion is done:

1. Convert bi-implications to implications
   $\alpha \iff \beta$ can be written as $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$

This also applies to compound sentences

$\alpha \iff (\beta \vee \gamma)$ can be written as $(\alpha \implies (\beta \vee \gamma)) \wedge ((\beta \vee \gamma) \implies \alpha)$

2. Convert implications to their logical equivalents

$\alpha \implies \beta$ can be written as $\neg\alpha \vee \beta$

3. Move negation inwards

CNF requires atomic literals. Hence, negation cannot appear on a compound statement. De Morgan's laws will be helpful here.

$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$

$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$

4. Distribute disjunction over conjunction

Disjunction and conjunction are distributive over each other. Now that we only have conjunctions, disjunctions and negations in our expression, we will distribute disjunctions over conjunctions wherever possible as this will give us a sentence which is a conjunction of simpler clauses, which is what we wanted in the first place. We need a term of the form

$(\alpha_1 \vee \alpha_2 \vee \alpha_3\ldots) \wedge (\beta_1 \vee \beta_2 \vee \beta_3\ldots) \wedge (\gamma_1 \vee \gamma_2 \vee \gamma_3\ldots)\wedge\ldots$

The `to_cnf` function executes this conversion using helper subroutines.

In [129…    `psource(to_cnf)`

```python
def to_cnf(s):
    """
    [Page 253]
    Convert a propositional logical sentence to conjunctive normal form.
    That is, to the form ((A | ~B | ...) & (B | C | ...) & ...)
    >>> 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` calls three subroutines. `eliminate_implications` converts bi-implications and implications to their logical equivalents.

`move_not_inwards` removes negations from compound statements and moves them inwards using De Morgan's laws.

`distribute_and_over_or` distributes disjunctions over conjunctions.

Run the cell below for implementation details.

In [130…   `psource(eliminate_implications)`
        `psource(move_not_inwards)`

```
psource(distribute_and_over_or)
```

```python
def eliminate_implications(s):
    """Change implications into equivalent form with only &, |, and ~ as logical operators."""
    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)


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)))
```

```python
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
```

Let's convert some sentences to see how it works

```python
In [131… A, B, C, D = expr('A, B, C, D')
         to_cnf(A |'<=>'| B)
```

```
Out[131… ((A | ~B) & (B | ~A))
```

```python
In [132… to_cnf(A |'<=>'| (B & C))
```

```
Out[132… ((A | ~B | ~C) & (B | ~A) & (C | ~A))
```

```python
In [133… to_cnf(A & (B | (C & D)))
```

```
Out[133… (A & (C | B) & (D | B))
```

```python
In [134… to_cnf((A |'<=>'| ~B) |'==>'| (C | ~D))
```

```
Out[134… ((B | ~A | C | ~D) & (A | ~A | C | ~D) & (B | ~B | C | ~D) & (A | ~B | C | ~D))
```

Coming back to our resolution problem, we can see how the `to_cnf` function is utilized here

```
In [135…   psource(pl_resolution)
```

```python
def pl_resolution(kb, alpha):
    """
    [Figure 7.12]
    Propositional-logic resolution: say if alpha follows from KB.
    >>> pl_resolution(horn_clauses_KB, A)
    True
    """
    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 [136…   pl_resolution(wumpus_kb, ~P11), pl_resolution(wumpus_kb, P11)
```

```
Out[136…   (True, False)
```

```
In [137…   pl_resolution(wumpus_kb, ~P22), pl_resolution(wumpus_kb, P22)
```

```
Out[137…   (False, False)
```

Forward and backward chaining

Previously, we said we will look at two algorithms to check if a sentence is entailed by the `KB` . Here's a third one. The difference here is that our goal now is to determine if a knowledge base of definite clauses entails a single proposition symbol q - the query. There is a catch however - the knowledge base can only contain **Horn clauses**.

Horn Clauses

Horn clauses can be defined as a *disjunction* of *literals* with **at most** one positive literal. A Horn clause with exactly one positive literal is called a *definite clause*. A Horn clause might look like

$\neg a \lor \neg b \lor \neg c \lor \neg d \ldots \lor z$

This, coincidentally, is also a definite clause.

Using De Morgan's laws, the example above can be simplified to

$a \wedge b \wedge c \wedge d \ldots \implies z$

This seems like a logical representation of how humans process known data and facts. Assuming percepts `a` , `b` , `c` , `d` ... to be true simultaneously, we can infer `z` to also be true at that point in time. There are some interesting aspects of Horn clauses that make algorithmic inference or *resolution* easier.

- Definite clauses can be written as implications:
  The most important simplification a definite clause provides is that it can be written as an implication. The premise (or the knowledge that leads to the implication) is a conjunction of positive literals. The conclusion (the implied statement) is also a positive literal. The sentence thus becomes easier to understand. The premise and the conclusion are conventionally called the *body* and the *head* respectively. A single positive literal is called a *fact*.
- Forward chaining and backward chaining can be used for inference from Horn clauses:
  Forward chaining is semantically identical to `AND-OR-Graph-Search` from the chapter on search algorithms. Implementational details will be explained shortly.
- Deciding entailment with Horn clauses is linear in size of the knowledge base:
  Surprisingly, the forward and backward chaining algorithms traverse each element of the knowledge base at most once, greatly simplifying the problem.

  The function `pl_fc_entails` implements forward chaining to see if a knowledge base `KB` entails a symbol `q` .
  Before we proceed further, note that `pl_fc_entails` doesn't use an ordinary `KB` instance. The knowledge base here is an instance of the `PropDefiniteKB` class, derived from the `PropKB` class, but modified to store definite clauses.
  The main point of difference arises in the inclusion of a helper method to `PropDefiniteKB` that returns a list of clauses in KB that have a given symbol `p` in their premise.

In [138...   `psource(PropDefiniteKB.clauses_with_premise)`

```python
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])]
```

Let's now have a look at the `pl_fc_entails` algorithm.

In [139...   `psource(pl_fc_entails)`

```python
def pl_fc_entails(kb, q):
    """
    [Figure 7.15]
    Use forward chaining to see if a PropDefiniteKB entails symbol q.
    >>> 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
```

The function accepts a knowledge base `KB` (an instance of `PropDefiniteKB`) and a query `q` as inputs.

`count` initially stores the number of symbols in the premise of each sentence in the knowledge base.
The `conjuncts` helper function separates a given sentence at conjunctions.
`inferred` is initialized as a *boolean* defaultdict. This will be used later to check if we have inferred all premises of each clause of the agenda.
`agenda` initially stores a list of clauses that the knowledge base knows to be true. The `is_prop_symbol` helper function checks if the given symbol is a valid propositional logic symbol.

We now iterate through `agenda`, popping a symbol `p` on each iteration. If the query `q` is the same as `p`, we know that entailment holds.
The agenda is processed, reducing `count` by one for each implication with a premise `p`. A conclusion is added to the agenda when `count` reaches zero. This means we know all the premises of that particular implication to be true.
`clauses_with_premise` is a helpful method of the `PropKB` class. It returns a list of clauses in the knowledge base that have `p` in their premise.

Now that we have an idea of how this function works, let's see a few examples of its usage, but we first need to define our knowledge base. We assume we know the following clauses to be true.

```
In [140…   clauses = ['(B & F)==>E',
                      '(A & E & F)==>G',
                      '(B & C)==>F',
                      '(A & B)==>D',
                      '(E & F)==>H',
                      '(H & I)==>J',
                      'A',
                      'B',
                      'C']
```

We will now `tell` this information to our knowledge base.

```
In [141…   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 [142…   pl_fc_entails(definite_clauses_KB, expr('G'))
```

Out[142…   True

```
In [143…   pl_fc_entails(definite_clauses_KB, expr('H'))
```

Out[143…   True

```
In [144…   pl_fc_entails(definite_clauses_KB, expr('I'))
```

Out[144…   False

```
In [145…   pl_fc_entails(definite_clauses_KB, expr('J'))
```

Out[145…   False

Effective Propositional Model Checking

The previous segments elucidate the algorithmic procedure for model checking. In this segment, we look at ways of making them computationally efficient.

The problem we are trying to solve is conventionally called the *propositional satisfiability problem*, abbreviated as the *SAT* problem. In layman terms, if there exists a model that satisfies a given Boolean formula, the formula is called satisfiable.

The SAT problem was the first problem to be proven *NP-complete*. The main characteristics of an NP-complete problem are:

- Given a solution to such a problem, it is easy to verify if the solution solves the problem.
- The time required to actually solve the problem using any known algorithm increases exponentially with respect to the size of the problem.

  Due to these properties, heuristic and approximational methods are often applied to find solutions to these problems.

It is extremely important to be able to solve large scale SAT problems efficiently because many combinatorial problems in computer science can be conveniently reduced to checking the satisfiability of a propositional sentence under some constraints.

We will introduce two new algorithms that perform propositional model checking in a computationally effective way.

1. DPLL (Davis-Putnam-Logeman-Loveland) algorithm

This algorithm is very similar to Backtracking-Search. It recursively enumerates possible models in a depth-first fashion with the following improvements over algorithms like `tt_entails`:

1. Early termination:
   In certain cases, the algorithm can detect the truth value of a statement using just a partially completed model. For example, $(P \vee Q) \wedge (P \vee R)$ is true if P is true, regardless of other variables. This reduces the search space significantly.

2. Pure symbol heuristic:
   A symbol that has the same sign (positive or negative) in all clauses is called a *pure symbol*. It isn't difficult to see that any satisfiable model will have the pure symbols assigned such that its parent clause becomes *true*. For example, $(P \vee \neg Q) \wedge (\neg Q \vee \neg R) \wedge (R \vee P)$ has P and Q as pure symbols and for the sentence to be true, P *has* to be true and Q *has* to be false. The pure symbol heuristic thus simplifies the problem a bit.

3. Unit clause heuristic:
   In the context of DPLL, clauses with just one literal and clauses with all but one *false* literals are called unit clauses. If a clause is a unit clause, it can only be satisfied by assigning the necessary value to make the last literal true. We have no other choice. Assigning one unit clause can create another unit clause. For example, when P is false, $(P \vee Q)$ becomes a unit clause, causing *true* to be assigned to Q. A series of forced assignments derived from previous unit clauses is called *unit propagation*. In this way, this heuristic simplifies the problem further.
   The algorithm often employs other tricks to scale up to large problems. However, these tricks are currently out of the scope of this notebook. Refer to section 7.6 of the book for more details.

   Let's have a look at the algorithm.

```
In [146...   psource(dpll)
```

```python
def dpll(clauses, symbols, model, branching_heuristic=no_branching_heuristic):
    """See if the clauses are true in a partial model."""
    unknown_clauses = []  # clauses with an unknown truth value
    for c in clauses:
        val = pl_true(c, model)
        if val is False:
            return False
        if val is None:
            unknown_clauses.append(c)
    if not unknown_clauses:
        return model
    P, value = find_pure_symbol(symbols, unknown_clauses)
    if P:
        return dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic)
    P, value = find_unit_clause(clauses, model)
    if P:
        return dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic)
    P, value = branching_heuristic(symbols, unknown_clauses)
    return (dpll(clauses, remove_all(P, symbols), extend(model, P, value), branching_heuristic) or
            dpll(clauses, remove_all(P, symbols), extend(model, P, not value), branching_heuristic))
```

The algorithm uses the ideas described above to check satisfiability of a sentence in propositional logic. It recursively calls itself, simplifying the problem at each step. It also uses helper functions `find_pure_symbol` and `find_unit_clause` to carry out steps 2 and 3 above.

The `dpll_satisfiable` helper function converts the input clauses to *conjunctive normal form* and calls the `dpll` function with the correct parameters.

In [147...   `psource(dpll_satisfiable)`

```python
def dpll_satisfiable(s, branching_heuristic=no_branching_heuristic):
    """Check satisfiability of a propositional sentence.
    This differs from the book code in two ways: (1) it returns a model
    rather than True when it succeeds; this is more useful. (2) The
    function find_pure_symbol is passed a list of unknown clauses, rather
    than a list of all clauses and the model; this is more efficient.
    >>> dpll_satisfiable(A |'<=>'| B) == {A: True, B: True}
    True
    """
    return dpll(conjuncts(to_cnf(s)), prop_symbols(s), {}, branching_heuristic)
```

Let's see a few examples of usage.

```
In [148…   A, B, C, D = expr('A, B, C, D')
```

```
In [149…   dpll_satisfiable(A & B & ~C & D)
```

```
Out[149…   {B: True, A: True, C: False, D: True}
```

This is a simple case to highlight that the algorithm actually works.

```
In [150…   dpll_satisfiable((A & B) | (C & ~A) | (B & ~D))
```

```
Out[150…   {B: True, C: True, D: False}
```

If a particular symbol isn't present in the solution, it means that the solution is independent of the value of that symbol. In this case, the solution is independent of A.

```
In [151…   dpll_satisfiable(A |'<=>'| B)
```

```
Out[151…   {B: True, A: True}
```

```
In [152…   dpll_satisfiable((A |'<=>'| B) |'==>'| (C & ~A))
```

```
Out[152…   {C: True, B: True, A: False}
```

```
In [153…   dpll_satisfiable((A | (B & C)) |'<=>'| ((A | B) & (A | C)))
```

```
Out[153…   {B: True, A: True}
```

2. WalkSAT algorithm

This algorithm is very similar to Hill climbing. On every iteration, the algorithm picks an unsatisfied clause and flips a symbol in the clause. This is similar to finding a neighboring state in the `hill_climbing` algorithm.
The symbol to be flipped is decided by an evaluation function that counts the number of unsatisfied clauses. Sometimes, symbols are also flipped randomly to avoid local optima. A subtle balance between greediness and randomness is required. Alternatively, some versions of the algorithm restart with a completely new random assignment if no solution has been found for too long as a way of getting out of local minima of numbers of unsatisfied clauses.

Let's have a look at the algorithm.

```
In [154…   psource(WalkSAT)
```

```python
def WalkSAT(clauses, p=0.5, max_flips=10000):
    """Checks for satisfiability of all clauses by randomly flipping values of variables
    >>> WalkSAT([A & ~A], 0.5, 100) is None
    True
    """
    # Set of all symbols in all clauses
    symbols = {sym for clause in clauses for sym in prop_symbols(clause)}
    # model is a random assignment of true/false to the symbols in clauses
    model = {s: random.choice([True, False]) for s in symbols}
    for i in range(max_flips):
        satisfied, unsatisfied = [], []
        for clause in clauses:
            (satisfied if pl_true(clause, model) else unsatisfied).append(clause)
        if not unsatisfied:  # if model satisfies all the clauses
            return model
        clause = random.choice(unsatisfied)
        if probability(p):
            sym = random.choice(list(prop_symbols(clause)))
        else:
            # Flip the symbol in clause that maximizes number of sat. clauses
            def sat_count(sym):
                # Return the the number of clauses satisfied after flipping the symbol.
                model[sym] = not model[sym]
                count = len([clause for clause in clauses if pl_true(clause, model)])
                model[sym] = not model[sym]
                return count

            sym = max(prop_symbols(clause), key=sat_count)
        model[sym] = not model[sym]
    # If no solution is found within the flip limit, we return failure
    return None
```

The function takes three arguments:

1. The `clauses` we want to satisfy.
2. The probability `p` of randomly changing a symbol.
3. The maximum number of flips (`max_flips`) the algorithm will run for. If the clauses
   are still unsatisfied, the algorithm returns `None` to denote failure.
   The algorithm is identical in concept to Hill climbing and the code isn't difficult to
   understand.

   Let's see a few examples of usage.

```python
In [155…    A, B, C, D = expr('A, B, C, D')
```

In [156… `WalkSAT([A, B, ~C, D], 0.5, 100)`

Out[156…   `{B: True, A: True, C: False, D: True}`

This is a simple case to show that the algorithm converges.

In [157… `WalkSAT([A & B, A & C], 0.5, 100)`

Out[157…   `{B: True, A: True, C: True}`

In [158… `WalkSAT([A & B, C & D, C & B], 0.5, 100)`

Out[158…   `{B: True, A: True, C: True, D: True}`

In [159… `WalkSAT([A & B, C | D, ~(D | B)], 0.5, 1000)`

This one doesn't give any output because WalkSAT did not find any model where these clauses hold. We can solve these clauses to see that they together form a contradiction and hence, it isn't supposed to have a solution.

One point of difference between this algorithm and the `dpll_satisfiable` algorithms is that both these algorithms take inputs differently. For WalkSAT to take complete sentences as input, we can write a helper function that converts the input sentence into conjunctive normal form and then calls WalkSAT with the list of conjuncts of the CNF form of the sentence.

In [160…
```
def WalkSAT_CNF(sentence, p=0.5, max_flips=10000):
    return WalkSAT(conjuncts(to_cnf(sentence)), 0, max_flips)
```

Now we can call `WalkSAT_CNF` and `DPLL_Satisfiable` with the same arguments.

In [161… `WalkSAT_CNF((A & B) | (C & ~A) | (B & ~D), 0.5, 1000)`

Out[161…   `{B: True, A: True, C: True, D: False}`

It works!
Notice that the solution generated by WalkSAT doesn't omit variables that the sentence doesn't depend upon. If the sentence is independent of a particular variable, the solution contains a random value for that variable because of the stochastic nature of the algorithm.

Let's compare the runtime of WalkSAT and DPLL for a few cases. We will use the `%%timeit` magic to do this.

In [162…
```
sentence_1 = A |'<=>'| B
sentence_2 = (A & B) | (C & ~A) | (B & ~D)
sentence_3 = (A | (B & C)) |'<=>'| ((A | B) & (A | C))
```

In [163…
```
%%timeit
dpll_satisfiable(sentence_1)
```

```
dpll_satisfiable(sentence_2)
dpll_satisfiable(sentence_3)
```

398 μs ± 12 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

In [164...
```
%%timeit
WalkSAT_CNF(sentence_1)
WalkSAT_CNF(sentence_2)
WalkSAT_CNF(sentence_3)
```

267 μs ± 3.75 μs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

On an average, for solvable cases, `WalkSAT` is quite faster than `dpll` because, for a small number of variables, `WalkSAT` can reduce the search space significantly. Results can be different for sentences with more symbols though. Feel free to play around with this to understand the trade-offs of these algorithms better.

SATPlan

In this section we show how to make plans by logical inference. The basic idea is very simple. It includes the following three steps:

1. Constuct a sentence that includes:
      A. A colection of assertions about the initial state.
      B. The successor-state axioms for all the possible actions at each time up to some maximum time t.
      C. The assertion that the goal is achieved at time t.
2. Present the whole sentence to a SAT solver.
3. Assuming a model is found, extract from the model those variables that represent actions and are assigned true. Together they represent a plan to achieve the goals.

Lets have a look at the algorithm

In [165...
```
psource(SAT_plan)
```

```python
def SAT_plan(init, transition, goal, t_max, SAT_solver=cdcl_satisfiable):
    """
    [Figure 7.22]
    Converts a planning problem to Satisfaction problem by translating it to a cnf sentence.
    >>> transition = {'A': {'Left': 'A', 'Right': 'B'}, 'B': {'Left': 'A', 'Right': 'C'}, 'C': {'Left': 'B', 'Right':
'C'}}
    >>> SAT_plan('A', transition, 'C', 1) is None
    True
    """

    # Functions used by SAT_plan
    def translate_to_SAT(init, transition, goal, time):
        clauses = []
        states = [state for state in transition]

        # Symbol claiming state s at time t
        state_counter = itertools.count()
        for s in states:
            for t in range(time + 1):
                state_sym[s, t] = Expr('S_{}'.format(next(state_counter)))

        # Add initial state axiom
        clauses.append(state_sym[init, 0])

        # Add goal state axiom
        clauses.append(state_sym[first(clause[0] for clause in state_sym
                        if set(conjuncts(clause[0])).issuperset(conjuncts(goal))), time]) \
            if isinstance(goal, Expr) else clauses.append(state_sym[goal, time])

        # All possible transitions
        transition_counter = itertools.count()
        for s in states:
            for action in transition[s]:
                s_ = transition[s][action]
                for t in range(time):
                    # Action 'action' taken from state 's' at time 't' to reach 's_'
                    action_sym[s, action, t] = Expr('T_{}'.format(next(transition_counter)))

                    # Change the state from s to s_
                    clauses.append(action_sym[s, action, t] | '==>' | state_sym[s, t])
                    clauses.append(action_sym[s, action, t] | '==>' | state_sym[s_, t + 1])

        # Allow only one state at any time
        for t in range(time + 1):
```

```python
            # must be a state at any time
            clauses.append(associate('|', [state_sym[s, t] for s in states]))

        for s in states:
            for s_ in states[states.index(s) + 1:]:
                # for each pair of states s, s_ only one is possible at time t
                clauses.append((~state_sym[s, t]) | (~state_sym[s_, t]))

    # Restrict to one transition per timestep
    for t in range(time):
        # list of possible transitions at time t
        transitions_t = [tr for tr in action_sym if tr[2] == t]

        # make sure at least one of the transitions happens
        clauses.append(associate('|', [action_sym[tr] for tr in transitions_t]))

        for tr in transitions_t:
            for tr_ in transitions_t[transitions_t.index(tr) + 1:]:
                # there cannot be two transitions tr and tr_ at time t
                clauses.append(~action_sym[tr] | ~action_sym[tr_])

    # Combine the clauses to form the cnf
    return associate('&', clauses)

def extract_solution(model):
    true_transitions = [t for t in action_sym if model[action_sym[t]]]
    # Sort transitions based on time, which is the 3rd element of the tuple
    true_transitions.sort(key=lambda x: x[2])
    return [action for s, action, time in true_transitions]

# Body of SAT_plan algorithm
for t in range(t_max + 1):
    # dictionaries to help extract the solution from model
    state_sym = {}
    action_sym = {}

    cnf = translate_to_SAT(init, transition, goal, t)
    model = SAT_solver(cnf)
    if model is not False:
        return extract_solution(model)
return None
```

Let's see few examples of its usage. First we define a transition and then call `SAT_plan`.

```
In [166…   transition = {'A': {'Left': 'A', 'Right': 'B'},
                        'B': {'Left': 'A', 'Right': 'C'},
                        'C': {'Left': 'B', 'Right': 'C'}}
```

```
print(SAT_plan('A', transition, 'C', 2))
print(SAT_plan('A', transition, 'B', 3))
print(SAT_plan('C', transition, 'A', 3))
```

```
['Right', 'Right']
['Right']
['Left', 'Left']
```

Let us do the same for another transition.

In [167…
```
transition = {(0, 0): {'Right': (0, 1), 'Down': (1, 0)},
              (0, 1): {'Left': (1, 0), 'Down': (1, 1)},
              (1, 0): {'Right': (1, 0), 'Up': (1, 0), 'Left': (1, 0), 'Down': (1,
              (1, 1): {'Left': (1, 0), 'Up': (0, 1)}}}


print(SAT_plan((0, 0), transition, (1, 1), 4))
```

```
['Right', 'Down']
```

First-Order Logic Knowledge Bases: `FolKB`

The class `FolKB` can be used to represent a knowledge base of First-order logic
sentences. You would initialize and use it the same way as you would for `PropKB` except
that the clauses are first-order definite clauses. We will see how to write such clauses to
create a database and query them in the following sections.

Criminal KB

In this section we create a `FolKB` based on the following paragraph.
*The law says that it is a crime for an American to sell weapons to hostile nations. The
country Nono, an enemy of America, has some missiles, and all of its missiles were sold to
it by Colonel West, who is American.*
The first step is to extract the facts and convert them into first-order definite clauses.
Extracting the facts from data alone is a challenging task. Fortunately, we have a small
paragraph and can do extraction and conversion manually. We'll store the clauses in list
aptly named `clauses`.

In [168…
```
clauses = []
```

"... it is a crime for an American to sell weapons to hostile nations"
The keywords to look for here are 'crime', 'American', 'sell', 'weapon' and 'hostile'. We use
predicate symbols to make meaning of them.

- `Criminal(x)` : `x` is a criminal
- `American(x)` : `x` is an American
- `Sells(x ,y, z)` : `x` sells `y` to `z`
- `Weapon(x)` : `x` is a weapon

- `Hostile(x)` : `x` is a hostile nation

Let us now combine them with appropriate variable naming to depict the meaning of the sentence. The criminal `x` is also the American `x` who sells weapon `y` to `z`, which is a hostile nation.

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \implies \text{Criminal}(x)$$

In [169... ```
           clauses.append(expr("(American(x) & Weapon(y) & Sells(x, y, z) & Hostile(z)) ==>
           ```

"*The country Nono, an enemy of America*"
We now know that Nono is an enemy of America. We represent these nations using the constant symbols `Nono` and `America` . the enemy relation is show using the predicate symbol `Enemy` .

$$\text{Enemy}(\text{Nono}, \text{America})$$

In [170... ```
           clauses.append(expr("Enemy(Nono, America)"))
           ```

"*Nono ... has some missiles*"
This states the existence of some missile which is owned by Nono.
$\exists x \text{Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$. We invoke existential instantiation to introduce a new constant `M1` which is the missile owned by Nono.

$$\text{Owns}(\text{Nono}, \text{M1}), \text{Missile}(\text{M1})$$

In [171...  ```
           clauses.append(expr("Owns(Nono, M1)"))
           clauses.append(expr("Missile(M1)"))
           ```

"*All of its missiles were sold to it by Colonel West*"
If Nono owns something and it classifies as a missile, then it was sold to Nono by West.

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \implies \text{Sells}(\text{West}, x, \text{Nono})$$

In [172... ```
           clauses.append(expr("(Missile(x) & Owns(Nono, x)) ==> Sells(West, x, Nono)"))
           ```

"*West, who is American*"
West is an American.

$$\text{American}(\text{West})$$

In [173...  ```
           clauses.append(expr("American(West)"))
           ```

We also know, from our understanding of language, that missiles are weapons and that an enemy of America counts as "hostile".

$$\text{Missile}(x) \implies \text{Weapon}(x), \text{Enemy}(x, \text{America}) \implies \text{Hostile}(x)$$

In [174... ```
           clauses.append(expr("Missile(x) ==> Weapon(x)"))
           clauses.append(expr("Enemy(x, America) ==> Hostile(x)"))
           ```

Now that we have converted the information into first-order definite clauses we can create our first-order logic knowledge base.

In [175…
```
crime_kb = FolKB(clauses)
```

The `subst` helper function substitutes variables with given values in first-order logic statements. This will be useful in later algorithms. It's implementation is quite simple and self-explanatory.

In [176…
```
psource(subst)
```

```python
def subst(s, x):
    """Substitute the substitution s into the expression x.
    >>> subst({x: 42, y:0}, F(x) + y)
    (F(42) + 0)
    """
    if isinstance(x, list):
        return [subst(s, xi) for xi in x]
    elif isinstance(x, tuple):
        return tuple([subst(s, xi) for xi in x])
    elif not isinstance(x, Expr):
        return x
    elif is_var_symbol(x.op):
        return s.get(x, x)
    else:
        return Expr(x.op, *[subst(s, arg) for arg in x.args])
```

Here's an example of how `subst` can be used.

In [177…
```
subst({x: expr('Nono'), y: expr('M1')}, expr('Owns(x, y)'))
```

Out[177…
```
Owns(Nono, M1)
```

Inference in First-Order Logic

In this section we look at a forward chaining and a backward chaining algorithm for `FolKB`. Both aforementioned algorithms rely on a process called **unification**, a key component of all first-order inference algorithms.

Unification

We sometimes require finding substitutions that make different logical expressions look identical. This process, called unification, is done by the `unify` algorithm. It takes as input two sentences and returns a *unifier* for them if one exists. A unifier is a dictionary which stores the substitutions required to make the two sentences identical. It does so by recursively unifying the components of a sentence, where the unification of a variable

symbol `var` with a constant symbol `Const` is the mapping `{var: Const}` . Let's look at a few examples.

```
In [178…   unify(expr('x'), 3)
```

```
Out[178…   {x: 3}
```

```
In [179…   unify(expr('A(x)'), expr('A(B)'))
```

```
Out[179…   {x: B}
```

```
In [180…   unify(expr('Cat(x) & Dog(Dobby)'), expr('Cat(Bella) & Dog(y)'))
```

```
Out[180…   {x: Bella, y: Dobby}
```

In cases where there is no possible substitution that unifies the two sentences the function return `None` .

```
In [181…   print(unify(expr('Cat(x)'), expr('Dog(Dobby)')))
```
```
None
```

We also need to take care we do not unintentionally use the same variable name. Unify treats them as a single variable which prevents it from taking multiple value.

```
In [182…   print(unify(expr('Cat(x) & Dog(Dobby)'), expr('Cat(Bella) & Dog(x)')))
```
```
None
```

Forward Chaining Algorithm

We consider the simple forward-chaining algorithm presented in *Figure 9.3*. We look at each rule in the knowledge base and see if the premises can be satisfied. This is done by finding a substitution which unifies each of the premise with a clause in the `KB` . If we are able to unify the premises, the conclusion (with the corresponding substitution) is added to the `KB` . This inferencing process is repeated until either the query can be answered or till no new sentences can be added. We test if the newly added clause unifies with the query in which case the substitution yielded by `unify` is an answer to the query. If we run out of sentences to infer, this means the query was a failure.

The function `fol_fc_ask` is a generator which yields all substitutions which validate the query.

```
In [183…   psource(fol_fc_ask)
```

```python
def fol_fc_ask(kb, alpha):
    """
    [Figure 9.3]
    A simple forward-chaining algorithm.
    """
    # TODO: improve efficiency
    kb_consts = list({c for clause in kb.clauses for c in constant_symbols(clause)})

    def enum_subst(p):
        query_vars = list({v for clause in p for v in variables(clause)})
        for assignment_list in itertools.product(kb_consts, repeat=len(query_vars)):
            theta = {x: y for x, y in zip(query_vars, assignment_list)}
            yield theta

    # check if we can answer without new inferences
    for q in kb.clauses:
        phi = unify_mm(q, alpha)
        if phi is not None:
            yield phi

    while True:
        new = []
        for rule in kb.clauses:
            p, q = parse_definite_clause(rule)
            for theta in enum_subst(p):
                if set(subst(theta, p)).issubset(set(kb.clauses)):
                    q_ = subst(theta, q)
                    if all([unify_mm(x, q_) is None for x in kb.clauses + new]):
                        new.append(q_)
                        phi = unify_mm(q_, alpha)
                        if phi is not None:
                            yield phi
        if not new:
            break
        for clause in new:
            kb.tell(clause)
    return None
```

Let's find out all the hostile nations. Note that we only told the `KB` that Nono was an enemy of America, not that it was hostile.

```python
In [184…   answer = fol_fc_ask(crime_kb, expr('Hostile(x)'))
           print(list(answer))
```

[{x: Nono}]

The generator returned a single substitution which says that Nono is a hostile nation. See how after adding another enemy nation the generator returns two substitutions.

In [185…
```python
crime_kb.tell(expr('Enemy(JaJa, America)'))
answer = fol_fc_ask(crime_kb, expr('Hostile(x)'))
print(list(answer))
```

[{x: Nono}, {x: JaJa}]

Note: `fol_fc_ask` makes changes to the `KB` by adding sentences to it.

Backward Chaining Algorithm

This algorithm works backward from the goal, chaining through rules to find known facts that support the proof. Suppose `goal` is the query we want to find the substitution for. We find rules of the form $\text{lhs} \implies \text{goal}$ in the `KB` and try to prove `lhs`. There may be multiple clauses in the `KB` which give multiple `lhs`. It is sufficient to prove only one of these. But to prove a `lhs` all the conjuncts in the `lhs` of the clause must be proved. This makes it similar to *And/Or* search.

OR

The *OR* part of the algorithm comes from our choice to select any clause of the form $\text{lhs} \implies \text{goal}$. Looking at all rules's `lhs` whose `rhs` unify with the `goal`, we yield a substitution which proves all the conjuncts in the `lhs`. We use `parse_definite_clause` to attain `lhs` and `rhs` from a clause of the form $\text{lhs} \implies \text{rhs}$. For atomic facts the `lhs` is an empty list.

In [186…
```python
psource(fol_bc_or)
```

```python
def fol_bc_or(kb, goal, theta):
    for rule in kb.fetch_rules_for_goal(goal):
        lhs, rhs = parse_definite_clause(standardize_variables(rule))
        for theta1 in fol_bc_and(kb, lhs, unify_mm(rhs, goal, theta)):
            yield theta1
```

AND

The *AND* corresponds to proving all the conjuncts in the `lhs`. We need to find a substitution which proves each *and* every clause in the list of conjuncts.

In [187…
```python
psource(fol_bc_and)
```

```python
def fol_bc_and(kb, goals, theta):
    if theta is None:
        pass
    elif not goals:
        yield theta
    else:
        first, rest = goals[0], goals[1:]
        for theta1 in fol_bc_or(kb, subst(theta, first), theta):
            for theta2 in fol_bc_and(kb, rest, theta1):
                yield theta2
```

Now the main function `fl_bc_ask` calls `fol_bc_or` with substitution initialized as empty. The `ask` method of `FolKB` uses `fol_bc_ask` and fetches the first substitution returned by the generator to answer query. Let's query the knowledge base we created from `clauses` to find hostile nations.

In [188…
```python
# Rebuild KB because running fol_fc_ask would add new facts to the KB
crime_kb = FolKB(clauses)
```

In [189…
```python
crime_kb.ask(expr('Hostile(x)'))
```