01 ILP とは何か

ILP は **論理プログラミング（Prolog）**と **機械学習**を掛け合わせた枠組みです。個別の観測（例）から、一般的な規則＝論理プログラムを **帰納**します。 統計的な学習と違い、結果が *人間に読める論理式* として出てくるのが最大の特徴です。

学習器に渡すのは次の **3 つの材料**です。今回の例（家族関係）で具体的に見てみましょう。

材料	記号	中身（DAUGHTER の例）
背景知識 Background	**B**	既に知っている事実：`parent/2`, `female/1`, `male/1`
正例 Positive	**E⁺**	成り立ってほしい例：`daughter(mary,ann)`, `daughter(eve,tom)`
負例 Negative	**E⁻**	成り立ってはいけない例：`daughter(tom,ann)`, `daughter(mary,tom)` …

実際のファイルはこう書きます（背景知識 `bk.pl` と 例 `exs.pl`）。

```
% ===== 背景知識 B =====
parent(ann, mary).   parent(ann, tom).
parent(tom, eve).    parent(tom, ian).
female(ann).  female(mary).  female(eve).
male(tom).    male(ian).

% ===== 訓練例 E =====
pos(daughter(mary, ann)).   % 正例 E+
pos(daughter(eve,  tom)).
neg(daughter(tom, ann)).    % 負例 E−
neg(daughter(mary, tom)).
```

02 ILP が探すもの ——「仮説 H」

ILP のゴールは、次の 2 条件を同時に満たす **仮説 H**（＝規則）を見つけることです。 `⊨` は「論理的に導ける（含意する）」を表す記号です。

求める仮説 H の条件

B ∧ H ⊨ **E⁺**　**完全性** —— 背景知識と仮説から、すべての正例を導ける

B ∧ H ⊭ E⁻ **無矛盾性** —— どの負例も導いてしまわない

学習された規則 `daughter(A,B):- parent(B,A), female(A).` で確かめると：

- **正例** `daughter(mary,ann)` → `parent(ann,mary)` ✓ かつ `female(mary)` ✓ ⇒ **導ける**
- **負例** `daughter(tom,ann)` → `female(tom)` は偽 ⇒ **導かない**

両条件を満たすので、この H が解になります。

03 演繹の「逆向き」が帰納

普段の Prolog は規則から個別の結論を導きます（演繹）。ILP はその矢印を逆に辿り、 例から規則そのものを組み立てます。

演繹 · DEDUCTION

規則 + 事実

↓

個別の結論

Prolog が普段やること。「規則があるから、この結論が言える」。

帰納 · INDUCTION

事実（多数の例）

↓

一般的な規則

ILP がやること。「これらの例が成り立つ以上、この規則があるはず」。

04 負例が規則の形を決める

これは今回の実験で実際に起きた、ILP の核心を示す出来事です。 最初は負例が「男性の例」だけだったため、Popper は *より単純だが一般的すぎる*規則で満点を取ってしまいました。

弱い負例だと

```
daughter(A,B) :- female(A), parent(B,_).
```

「A が女性で、B が"誰かの"親」でも成立 —— parent と female が別人でよい

「女性だが娘ではない」負例 **neg(daughter(mary, tom))** を追加 ↓

正しい規則へ

```
daughter(A,B) :- parent(B,A), female(A).
```

parent と female が**同じ人物 A** で繋がる —— 本来の「娘」の定義

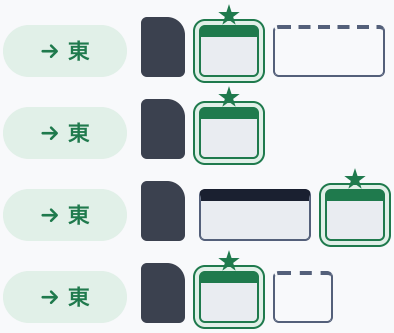
負例は単なる「ダメな例」ではなく、**仮説を絞り込む制約**です。 良い負例を選ぶことが、意図した規則を引き出す鍵になります。

05 もう少し大きな例 —— 東西列車問題

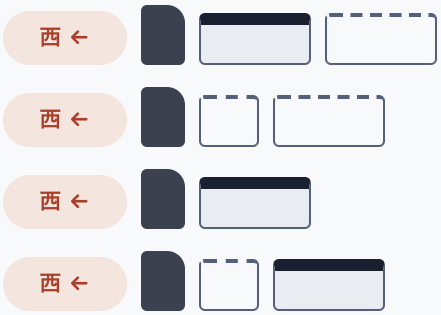
ILP の代名詞、Michalski の **East-West Trains**。8 本の列車があり、 東行き（4本）と西行き（4本）に分かれています。列車ごとに**車両の数はバラバラ**で、各車両は「短い／長い」「屋根が閉／開」という属性を持ちます。 **なぜ東行きなのか？** —— その規則を ILP に発見させます。

閉（屋根あり）　開（屋根なし）　幅が狭い＝短い / 広い＝長い　★ = 手がかりの車両

東行き（正例 E⁺）



西行き（負例 E⁻）



単純な特徴ひとつでは分けられない点に注意してください。「短い車両を持つ」だけでは西行きにも当てはまり、「閉じた車両を持つ」だけでも西行きに当てはまります。**2つの属性を同じ車両で組み合わせて** はじめて分離できます ── まさに ILP が関係の中から探し当てるものです。

POPPER が 8 本の列車だけから学習した規則

```
eastbound(T) :- has_car(T,C), short(C), closed(C).
```

「**短くて屋根の閉じた車両** C を持つ列車 T は東行き」── 図の ★ がまさにその車両

06 一般の機械学習と何が違うのか

同じ分類問題をニューラルネットなどで解くこともできます。しかし ILP は、**答えの形**そのものが違います。

観点	一般的な機械学習（例：ニューラルネット）	ILP
説明可能性	学習結果は大量の重みの数値。「なぜ東行きと判定したか」は不透明（ブラックボックス）。	規則そのものが答え。`eastbound(T):- has_car(T,C),short(C),closed(C).` と**そのまま読める**。
入力の形	固定長の特徴ベクトル。 列車ごとに車両数が違うと表現しにくい。	`has_car` という**関係**で自然に表現。 可変個の車両や構造をそのまま扱える。

観点	一般的な機械学習（例：ニューラルネット）	ILP
背景知識	主に人手の特徴量設計を通じて間接的に。	既知の**述語・規則を直接投入**できる。 ドメイン知識がそのまま探索の材料になる。
必要データ量	一般に大量の例が必要。	今回は**わずか 8 例**で正しい規則に到達。
再帰・関係	基本は非対応。	`ancestor` のように**自分自身を使う再帰規則**も帰納できる。

要するに ILP は、「**予測できる」だけでなく「なぜそう言えるかを規則として差し出す**」。 背景知識を持つ領域（医療・化学・法務・故障診断など）で、少ないデータから 検証可能な仮説を得たいときに特に力を発揮します。

手元で動かす（POPPER）

```
source ~/ILP/popper-venv/bin/activate
cd ~/ILP/Popper
python popper.py ~/ILP/daughter    # => daughter(A,B):- parent(B,A), female(A).
python popper.py ~/ILP/trains_ew   # => eastbound(T):- has_car(T,C), short(C),
# どちらも Precision 1.00 / Recall 1.00
```

07 応用：BCP —— ILP と機械学習を「つなぐ」

BCP（Bottom Clause Propositionalization／最下節命題化）は、 ILP とニューラルネットを橋渡しする手法です（França・Zaverucha・Garcez, 2014、システム名 **CILP++**）。 ⑥ では ILP と一般の機械学習を対比しましたが、BCP は**その両方の良いとこ取り**を狙います。

> **純粋な ILP との違い。** Popper のような ILP は「論理プログラムの空間」を直接探索して規則を作ります。 BCP は規則を直接探さず、まず各例を **最下節（bottom clause）**経由で**特徴**

ベクトルに変換（命題化）し、 あとは高速・頑健な**命題学習器（ニューラルネット等）**に任せます。 いわば *ILP のフロントエンド × 機械学習のバックエンド*。

STEP 1

最下節を構築

各例を、背景知識をたどって「最も具体的な節」に展開する。関係情報がここに畳み込まれる。

→

STEP 2

命題化

最下節のリテラルを特徴に読み替え、各例を 0/1 の特徴ベクトルにする。

→

STEP 3

命題学習器（NN 等）

ニューラルネット・SVM など何でも後段に。速く、ノイズに強く、重みで説明性も残る。

簡単な例題 —— 東西列車を BCP で解く

⑤ と同じ列車データを使います。まず 1 本の列車を最下節に展開します。

STEP 1 — 例 EASTBOUND(T1) の最下節（背景知識をたどって構築）

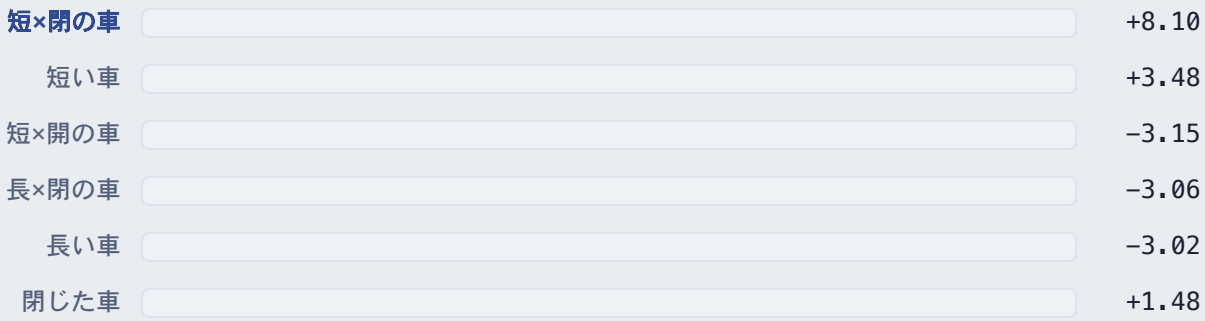
```
eastbound(A) :- has_car(A,B), short(B), closed(B), has_car(A,C)
```

STEP 2。最下節のリテラルから関係的な特徴を作り、各列車を 0/1 ベクトルにします （紙面の都合で代表的な列だけ表示）。

列車	短い車	閉じた車	短×閉の車	東行き
t1	1	1	1	東
t2	1	1	1	東
t3	1	1	1	東
t4	1	1	1	東
t5	0	1	0	西
t6	1	0	0	西
t7	0	1	0	西
t8	1	1	0	西

「短い車を持つ」だけ、「閉じた車を持つ」だけでは東西を分けられません（西行きにも 1 が混ざる）。 ところが背景知識の関係から作った **「短くて閉じた車を持つ」特徴**は、東行きの列と**ぴったり一致**します。

STEP 3。この表を命題学習器（ここでは 1 層ニューラルネット＝ロジスティック回帰）に学習させると **正解率 100%**。学習された重みは、どの関係的特徴が効いたかを示します。



最大の重みは「短×閉の車」—— ⑤ で ILP が見つけた規則の本体 `has_car(T,C),short(C),closed(C)` が、そのまま効く特徴として浮かび上がる。

BCP の利点

- **速い・スケールする** —— 一階述語の探索を避け、成熟した命題学習器に載せられる。大規模な関係データに強い。
- **ノイズに強い** —— 統計的・ニューラルな学習器の頑健さを継承（例外や誤りに弱い古典 ILP を補う）。
- **道具箱がそのまま使える** —— 後段はニューラルネット・SVM・決定木など何でも。深層学習とも接続できる。
- **背景知識を保持** —— 最下節を経由するので関係情報が特徴に入る。単なる手作りの特徴量設計より豊か。
- **説明性が残る** —— 重みが「どの関係的特徴が効いたか」を指す（この例では短×閉）。完全なブラックボックスにならない。

トレードオフは、出力が ⑤ のような**単一の簡潔な論理規則ほどは澄んでいない**こと、 最下節が大きくなり得ること。BCP は **純 ILP（Popper）と命題的な機械学習の“中間”**に位置し、 背景知識を活かしつつ大規模・ノイズありのデータを速くさばきたい場面に向きます。

手元で動かす（自作の最小 BCP）

```
source ~/ILP/popper-venv/bin/activate
python ~/ILP/bcp/bcp_trains.py
# 最下節 → 特徴表 → 1層NN学習 → 正解率100%、重み最大= 短×閉
```

08 もうひとつの応用：LFIT —— システムの「動き方」を学ぶ

LFIT（Learning From Interpretation Transition）は、井上克己・Ribeiro・Sakama が提唱した枠組みです（*Machine Learning*, 2014）。「LIFT」と呼ばれることもありますが同じもの。これまで（①〜⑦）は「これは娘か？ 東行きか？」という**静的な分類規則**を学びました。LFIT が学ぶのは違います。

> **何が違うのか。** LFIT は**時間とともに変化するシステムの“動力学”**を学びます。入力は「いまの状態 → 次の状態」という**状態遷移の対**。出力は、その遷移を再現する **正規論理プログラム（＝システムの規則）**。ラベル付き例ではなく**観測された振る舞い**から、世界がどう動くのかを逆算します。

簡単な例題 —— 2 変数の振動子

変数 `p`, `q` だけの小さなシステムを観測したら、状態が次のように ぐるぐる巡回していたとします（`∅`＝どちらも偽、`{p,q}`＝両方真）。**規則は分かりません。観測された遷移だけ**が手元にあります。

`∅` → `{q}` → `{p,q}` → `{p}` → ↻ ∅ に戻る（周期 4）

この 4 つの遷移だけを LFIT に与えると、動力学の規則を復元します。

LFIT が状態遷移だけから学習した規則（`'` は「次の時刻」）

```
p' :- q.
```

```
q' :- not p.
```

「次の **p** は、いまの **q** が真なら真」／「次の **q** は、いまの **p** が偽なら真」—— この 2 規則を回すと、上の巡回がぴったり再現される。

観測（振る舞い）から、その裏で働く**仕組みそのもの**を読める論理で取り出せました。 同じやり方で、遺伝子制御ネットワークやセルオートマトンのルールを、観測データだけから発見できます。

メリット・デメリット

メリット

- **動力学が読める** —— モデルが人間に理解できる論理プログラムとして出る（説明可能な力学モデル）。
- **モデル未知でも学べる** —— 方程式や回路を知らなくても、観測された遷移だけから逆算できる。
- **応用が広い** —— 遺伝子制御・ブーリアンネットワーク・セルオートマトン・システム生物学。
- **拡張が豊富** —— 多値・遅延（LFkT）・確率的・非同期版があり、逐次／anytime 学習も可能。

デメリット

- **状態爆発** —— 変数 n 個で状態は 2^n。完全な復元には多数の遷移が要る（スケーラビリティ）。
- **理想化の前提** —— 古典版は同期・決定的・ノイズなしを仮定。ノイズや非同期は拡張が必要。
- **全観測が前提** —— すべての変数が観測できる想定。隠れ変数があると弱い。
- **規則が増え得る** —— 複雑な系では規則数・条件が大きくなり、可読性が下がることも。

手元で動かす（自作の LFIT 実装）

```
source ~/ILP/popper-venv/bin/activate
cd ~/ILP/lfit
python demo_boolean_network.py
# 状態遷移だけから元のブーリアンネットワークを復元し、全遷移の再現を検証
```

09 発展：セルオートマトンのルールを「見て」学ぶ

LFIT の力がはっきり出る例をもう一つ。下の三角形は **Rule 90** という セルオートマトンが、中央 1 個の点から時間発展した様子です（有名な Sierpinski 三角形）。 各セルは毎ステップ、単純な局所ルールで 0/1 を更新しています。

↓ 時間発展（上が初期、下ほど後の時刻）— このパターンを生む「規則」を LFIT に当てさせる

この振る舞いだけ —— 具体的には **5 セルをリング状に並べた系の全 32 状態の遷移** —— を LFIT に見せます。ルールは一切教えません。すると各セルの更新規則を復元します。

LFIT が遷移だけから学習した規則（全セル分）

```
% 次の値 = 左隣 XOR 右隣
c0' :- c1, not c4.
c0' :- c4, not c1.
c1' :- c0, not c2.
c1' :- c2, not c0.
c2' :- c1, not c3.
c2' :- c3, not c1.
c3' :- c2, not c4.
c3' :- c4, not c2.
c4' :- c0, not c3.
c4' :- c3, not c0.
```

どのセルも同じ形 —— **「左隣が真で右隣が偽」または「左隣が偽で右隣が真」**のとき次に真、 つまり **左隣 XOR 右隣**。これはまさに Rule 90 の定義そのもの。

つまり LFIT は、**系の“進化を眺めているだけ”で、その背後で働く物理法則（局所ルール）を 読める論理として取り出しました。**同じ枠組みで、遺伝子ネットワークの制御則や 未知のダイナミクスを、観測データから発見できるわけです。

> **補足。** ここでは命題版なので「セルごと」に同型の規則を復元しました。一階版の LFIT を使えば、 `cell(X)` と隣接関係を導入して **位置に依らない 1 本の一般規則**として学習することもできます。

10 一階版 LFIT —— Rule 90 を「1つの一般規則」に

⑨ の命題版は、セルごとに 10 本の規則を復元しました（ `c0'…c4'` ）。同じ形の規則が 位置の数だけ並んでいた —— これは「位置」を扱えていないからです。 **一階（first-order）版**では、位置を**引数**にすることで、 これを**たった 1 組の一般規則**にまとめます。

> **表現の変え方。** 命題 `c0,c1,…` の代わりに、状態 S における各セルの値を `on(Cell,S)` / `off(Cell,S)` で表し、リングの位相を静的な関係 `left(X,L)` / `right(X,R)` で与える。あとは本格 ILP システム **Popper** に、全 32 状態の遷移を例として学習させます。

POPPER が学習した一階規則（位置 X に依らない）

```
nexton(X,S) :- left(X,L), on(L,S), right(X,R), off(R
nexton(X,S) :- left(X,L), off(L,S), right(X,R), on(R
```

「セル X は、**左隣 L と右隣 R のうち片方だけが on** のとき、次に on」—— すべてのセルを 1 組で言い切る、位置に依らない Rule 90（＝左隣 XOR 右隣）。

⑨ 命題版

⑩ 一階版

規則 10 本

.

c0…c4 に固定

N=5 専用

位置ごとに同型の規則が並ぶ。セル数が変わると作り直し。

規則 2 本

.

位置 X を変数化

任意のリングに適用

1 組の一般規則。**学習していないセル数のリングにもそのまま通用**する。

これが一階 ILP の威力です。**「位置を変数にする」だけで、無数の個別ケースが 1 つの法則に凝縮**され、 見たことのない規模へ一般化できます。LFIT × 一階表現は、空間的・関係的なダイナミクスを コンパクトで転移可能な論理として取り出す —— ここまでの全要素（説明可能性・背景知識・関係・動力学）が 合流する到達点です。

手元で動かす（データ生成 → POPPER で一階学習）

```
source ~/ILP/popper-venv/bin/activate
python ~/ILP/lfit_fo/generate.py        # 全32遷移を on/off + left/right で生成
cd ~/ILP/Popper && python popper.py ~/ILP/lfit_fo
# => nexton(X,S):- left(X,L),on(L,S),right(X,R),off(R,S).  他1本(Precision/Recal
```

11 Progol と逆伴意 —— 事例を「言い替えて」仮説を探す

ここまでの ⑥–⑩ は **Popper（LFF：生成→検査→制約）**でした。最後に、ILP のもう一方の源流 —— **Progol の逆伴意（inverse entailment）** —— を、東京理科大・滝本宗宏先生の講義 「論理型人工知能入門 —安全安心な人工知能を目指して—」の 題材 `animals.pl`（動物を `mammal / fish / reptile / bird` に分類）で実装・再現します。 Progol は Muggleton による古典的 ILP システムです。

> **逆伴意の考え方。** 背景知識 B と正例 e に対し、B ∧ H ⊨ e となる仮説 H がほしい。これを移項すると B ∧ ¬e ⊨ ¬H。そこで **まず ¬H に当たる 最も具体的な節 = 最飽和節（MSH,**

> **⊥）**を、**事例を背景知識で「言い替えて」**作り、 次に **⊥ を下界とする一般化の束を探索**して正例・負例で検査する。滝本先生の資料 （*MSH の生成 → 仮説の探索 → 事例検査*）の流れそのものです。

Progol は各述語の **モード宣言**で探索語彙を与えます（`animals.pl` より）：

```
% 頭部:class(+animal,#class)  ── #class は学習される定数(mammal/fish/…)
:- modeh(1, class(+animal,#class))?
:- modeb(1, has_milk(+animal))?
:- modeb(1, has_gills(+animal))?
:- modeb(1, has_covering(+animal,#covering))?
:- modeb(1, has_legs(+animal,#nat))?
:- modeb(1, homeothermic(+animal))?
:- modeb(*, habitat(+animal,#habitat))?
```

正例 `class(dolphin,mammal)` を背景知識で言い替えると、イルカについて言える事実を すべて集めた **最飽和節** ⊥ ができます：

最飽和節 MSH（⊥）—— 1 例を背景知識で飽和して変数化

```
class(A,mammal) :- has_milk(A), homeothermic(A), has
```

イルカ 1 頭から作った「最も具体的な仮説」。探索はこの ⊥ を**下界**とし、 余分な条件を落として一般化しながら、正例を覆い負例を外す最小の節を探します。

これを本ページの **Popper（LFF）**で実装・実行しても、⊥ の中から **不要な条件が削ぎ落とされ**、各クラスにつき次の 1 本が学習されます （16 頭の例＋背景知識、いずれも Precision / Recall ＝ 1.00）：

ILP が学習した動物分類規則（POPPER で再現）

```
mammal(A) :- has_milk(A).
fish(A) :- has_gills(A).
reptile(A) :- has_scales(A), no_gills(A).
```

```
bird(A) :- has_feathers(A).
```

「乳を出す＝哺乳類」「えらがある＝魚類」「鱗がありえらが無い＝爬虫類」「羽毛がある＝鳥類」。 例と背景知識だけから、教科書的な規則が**言葉のまま**出てくる。

PROGOL（逆伴意）

例を飽和 → ⊥

.

⊥ を下界に

一般→特殊へ探索

1 例から最具体の ⊥ を作り、その部分節だけを調べる。モード宣言で空間を絞る。

POPPER（LFF）

仮説を生成

.

検査 → 失敗を

制約に変える

ASP で候補を生成し、失敗を制約に変えて枝刈り。本ページ ⑥–⑩ の方式。

注目すべきは、出てくるのが **重み行列ではなく「読める規則」**だという点です。 なぜその分類かを人が検証でき、背景知識も差し替えられる —— これが滝本先生の言う **「安全・安心な AI」**そのものです。深層学習との橋渡し（ニューロ記号処理）は 本ページ ⑦ の **BCP** が受け持ちます。

手元で動かす（ANIMALS.PL を POPPER へ移植して学習）

```
source ~/ILP/popper-venv/bin/activate
python ~/ILP/animals/gen.py            # mune/ai の animals.pl → Popper タスク4種
for c in mammal fish reptile bird; do
  python ~/ILP/Popper/popper.py ~/ILP/animals/$c
done
# => mammal(A):-has_milk(A). / fish(A):-has_gills(A). /
#    reptile(A):-has_scales(A),no_gills(A). / bird(A):-has_feathers(A).
```

サンプル一式：~/ILP/daughter/（入門）、~/ILP/trains_ew/（東西列車）、 ~/ILP/animals/（動物分類・Progol の animals.pl を移植）、 ~/ILP/bcp/bcp_trains.py（BCP の最小実装）、~/ILP/lfit/（LFIT の実装、 demo_boolean_network.py / demo_ca.py）、~/ILP/lfit_fo/（一階版 LFIT）。 ILP 学習器は Popper 5.0.0（LFF 方式）、BCP・LFIT は本ページ用の自作実装。 参考：Muggleton (1995) *Inverse*

Entailment and Progol、滝本宗宏「論理型人工知能入門」（`animals.pl` の出典）、 França+ (2014) *BCP / CILP++*、Inoue+ (2014) *Learning From Interpretation Transition*。