The Game Description Language (GDL) Part 1

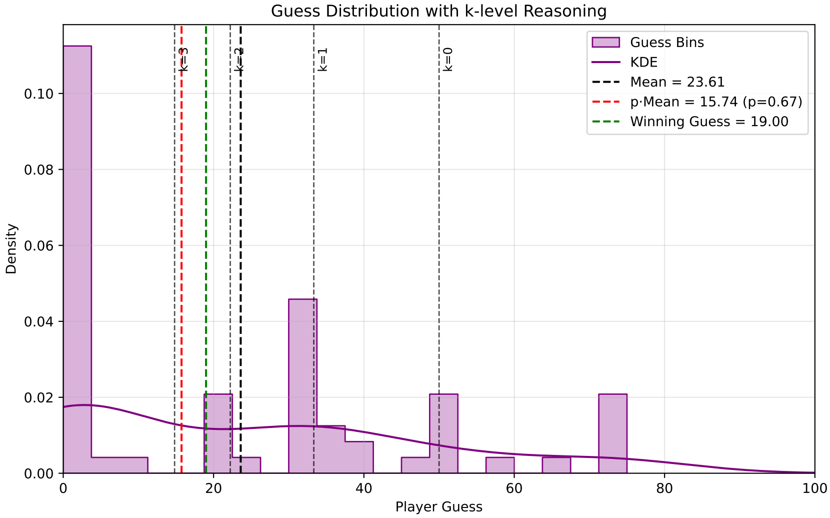
Agnieszka Mensfelt **Kostas Stathis** **Vince Trencsenyi**

https://dicelab-rhul.github.io/Strategic-AI-Autoformalization

ESSAI, 01/07/25

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Game Results



Outline

What is GDL

What is GDL?

The **Game Description Language**[1] **(GDL)** is a formal language used to define the rules of any game in a machine-readable format.

- It is based on **logic programming**, using a Datalog-style syntax.

Why GDL?

- Allows AI agents to **read and interpret new games** without manual reprogramming.
- Considered as the foundation for **General Game Playing**[2] **(GGP)**.

[1] N. Love, M. Genesereth, and T. Hinrichs, "General game playing: Game description language specification," Stanford University, Logic Group, Technical Report LG-2006-01, 2006.

[2] M. R. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition," *AI Mag.*, vol. 26, no. 2, pp. 62–72, 2005.

Digression on Logic Programs

A **logic program**[3] is a set of Horn clauses, each written as:

$$A_0 \leftarrow A_1 \wedge A_2 \wedge \cdots \wedge A_n \quad (\text{where } n \geqslant 0)$$

- Each A_i is an **atomic formula** $p(t_1, \ldots, t_m)$:
  - p is a predicate symbol
  - t_i are **terms** (constants, variables, or $f(t_1, \ldots, t_m)$)
  - f is a function symbol
- Variables in terms are **universally quantified**, and the scope is the clause in which the variable occurs
- Notation:
  - $\leftarrow$ means "if"
  - $\wedge$ means "and"

[3] R. A. Kowalski, "Logic programming," in *Computational Logic*, ser. Handbook of the History of Logic, J. H. Siekmann, Ed., vol. 9, Elsevier, 2014, pp. 523–569.

Logic Programs

Clause Types

- In a clause:

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_n$$

  - A_0 is called the **head** (or conclusion)
  - $A_1 \wedge \cdots \wedge A_n$ is the **body** (or conditions)
- If $n > 0$, the clause is a **rule**. E.g.: Birds fly is expressed as

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X)$$

- Recall: Variables in clauses are implicitly universally quantified

$$\forall X(\mathit{flies}(X) \leftarrow \mathit{bird}(X))$$

  but the quantifiers are omitted.

Logic Programs

Clause Types (cnt'd)

- If $n = 0$, the body is equivalent to `true`, and the clause becomes a **fact**:

$$A_0 \leftarrow \text{true} \quad \text{is abbreviated as } A_0$$

E.g.: To state the fact that 'tweety is a bird', we write

$$bird(tweety) \leftarrow true$$

or simply

$$bird(tweety)$$

- If the head A_0 is `false`, the clause is a **goal clause**, abbreviated as:

$$\leftarrow A_1 \wedge \cdots \wedge A_n$$

That is, deny that $A_1 \wedge \cdots \wedge A_n$ has a solution, and refute it by finding one. E.g.: Here is a goal: 'Does tweety fly'? We deny it:

$$\leftarrow \mathit{fly}(\mathit{tweety})$$

and then refute the denial using

$$(\mathit{flies}(X) \leftarrow \mathit{bird}(X) \text{ and } \mathit{bird}(\mathit{tweety})) \text{ and } \theta = \{X/\mathit{tweety}\}$$

thus confirming that is true, from what we know about birds so far.

Normal Logic Programs

Horn clauses fail to capture **non-monotonic reasoning**, so we extend them to clauses of the form:

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_n \wedge \text{not } B_1 \wedge \cdots \wedge \text{not } B_m \quad (\text{where } n \geqslant 0 \text{ and } m \geqslant 0)$$

- Each A_i and B_j is an **atomic formula**.
- not is interpreted as **negation by failure (NBF)**.
- Atomic formulas and their negations are called **literals**:
  - A_i are **positive literals**
  - *not* B_j are **negative literals**
- Clause sets are called **normal logic programs** (or *logic programs*).

Normal Logic Programs

Example of non-monotonicity

Flying birds revisited as a normal logic program:

$$\mathit{flies}(X) \leftarrow \mathit{bird}(X) \wedge \ \mathit{not}\ \ \mathit{penguin}(X)$$

- If we only know

  $$\mathit{bird}(\mathit{tweety})$$

  then we can conclude that by default tweety can fly.
- If we later discover that tweety is also a penguin:

  $$\mathit{penguin}(\mathit{tweety})$$

  then we can conclude that tweety cannot fly.

Normal Logic Programs: Example of Non-Monotonicity (cont'd)

More general (methodological) formulations are possible. For example:

$$\text{flies}(X) \leftarrow \text{bird}(X) \wedge \text{not abnormal\_bird}(X)$$

- If we only know:

$$\text{bird}(\text{tweety})$$

  then we can conclude that, by default, tweety can fly.
- If we later discover that penguins do not fly and tweety is a penguin, we write:

$$\text{abnormal\_bird}(X) \leftarrow \text{penguin}(X)$$

$$\text{penguin}(\text{tweety})$$

  then we can conclude that tweety cannot fly (and similarly with ostriches,....)

GDL - Syntax, formalism and examples

Syntactic Variations of GDL

Recall notation for normal logic programs :

$$A_0 \leftarrow A_1 \wedge \cdots \wedge A_n \wedge \text{not } B_1 \wedge \cdots \wedge \text{not } B_m$$

GDL is normally presented in Lisp-like[4] and Prolog-like[5] notations:

Lisp-like Prefix Notation

(<= A_0 A_1 ... A_n (not B_1) ... (not B_m))

E.g.

(<= (flies ?x) (bird ?x)(not (penguin ?x)))

Prolog-like Infix Notation

A_0 :- A_1, ..., A_n, not B_1, ... , not B_m.

E.g.:

flies(X) :- bird(X), not penguin(X).

[4] R. Jones, C. Maynard, and I. Stewart, *The Art of Lisp Programming*. Springer London, 1990.

[5] L. S. Sterling and E. Y. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, Second. MIT Press, 1994.

GDL Specification Process for Games

Question: How do we specify games in GDL?

- GDL defines a **shared vocabulary** to support game representation:
  - Establishes a fixed, **game-independent vocabulary** - shared meaning for key terms across all games.
  - Allows use of **game-specific vocabulary** - authors can define their own predicates and constants for individual games.
- Includes **predefined object constants** (0 to 100):
  - Often used to define **utility values** for game outcomes (e.g., $0 =$ worst, $100 =$ best)

Game Independent Vocabulary

`Core Vocabulary` [6]	**Explanation**
`role(R)`	R is a role in the game.
`input(R, A)`	A is a feasible action for role R.
`base(F)`	F is a base proposition in the game.
`init(F)`	Fact F is true at the beginning of the game.
`true(F)`	Fact F is true in the current state.
`does(R, A)`	Action A taken by role R.
`legal(R, A)`	Action A is legal for R in the current state.
`next(F)`	Fact F is true in the next state.
`goal(R, U)`	Role R gets payoff U in the current state.
`terminal`	Declares terminal states.

Represent a game in terms of this vocabulary, plus any additional predicates.

[6] We omit distinct(X,Y) stating $X \neq Y$.

PD as a concrete GDL Formalization

We illustrate next how to instantiate the game independent vocabulary by formalizing a specific game:

- We will use the **Prisoner's Dilemma** (PD) as our example.
- We have already discussed PD previously.

	C	D
C	(3, 3)	(0, 5)
D	(5, 0)	(1, 1)

- The goal is to see how to represent PD in GDL (not to run it).

PD Formalization: Roles, Base and Inputs

Defined Roles:

```
% --- Player roles: row and col
% --- from the matrix

    role(row).
    role(col).
```

Base:

```
% --- Base propositions for PD ---

    base(control(R)) :- role(R).
    base(did(R, A) :- role(R), action(A).

    action(coop).
    action(defect).
```

PD Formalization: Inputs and Initial State


```
\textbf{Inputs:}
% --- Feasible actions for roles

    input(R, A):- role(R), action(A).
```

Initial State:

```
% --- What holds initially ---

    init(control(row)).
    init(control(col)).
```

PD Formalization: Legal Moves and State Transitions

Legal Moves:

```
% --- legal moves ---

    legal(R, A) :- input(R, A), true(control(R)).
```

State Transitions:

```
% --- state transitions ---

    next(did(R, M)) :- does(R, M), true(control(R)).
```

PD Formalization: Payoff Rules


```
% --- Both cooperate ---

goal(R1, 3) :-
    true(did(R1, coop)),
    true(did(R2, coop)),
    distinct(R1, R2).

% --- Both defect ---

goal(R1, 1) :-
    true(did(R1, defect)),
    true(did(R2, defect)),
    distinct(R1, R2).
```

```
% --- Sucker's Payoff ---

goal(R1, 0) :-
    true(did(R1, coop)),
    true(did(R2, defect)),
    distinct(R1, R2).

% --- Temptation Payoff ---

goal(R1, 5) :-
    true(did(R1, defect)),
    true(did(R2, coop)),
    distinct(R1, R2).
```

PD Formalization: Termination Condition


```
% --- Termination ---

terminal :-
    true(did(R1, M1)),
    true(did(R2, M2)),
    distinct(R1, R2).
```

- This rule defines when the game ends: both players have made a move.
- We have now completed the formalization of the Prisoner's Dilemma.
- Emphasis has been on **declarative representation**, not execution.

Question:
What would a `Datalog`-style reasoning engine derive from this?

PD State Transitions (GDL Trace)

How should the specification work?

- Reason forwards from initial state until terminal state is reached[7].

Initially
```
init(control(row))
init(control(col))
```

apply rules →

Before Moves
```
true(control(row))
true(control(col))
legal(row, coop)
legal(col, coop)
legal(row, defect)
legal(col, defect)
```

+

Moves Dataset
```
does(row, coop)
does(col, defect)
```

transition update →

After Moves
```
did(row, coop)
did(col, defect)
goal(row, 1)
goal(col, 5)
```

- We assume a *Game Manager* for state and action coordination.

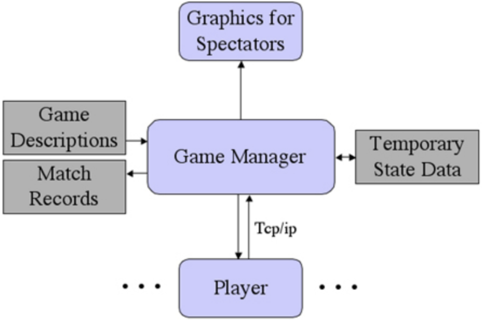
[7] M. R. Genesereth and M. Thielscher, *General Game Playing* (Synthesis Lectures on Artificial Intelligence and Machine Learning). Morgan & Claypool Publishers, 2014.

Game Manager

Game Manager Behaviour

GGP manager role

- **Game Manager**:
  - maintains a database of game descriptions and match records;
  - maintains temporary state for matches in progress;
  - communicates with game players;
  - provides a user interface for scheduling matches; and
  - provides graphics for spectators watching ongoing matches.

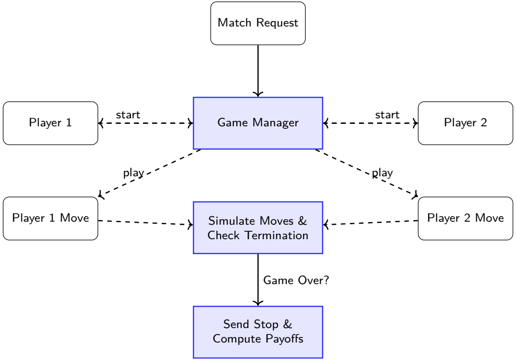


General Game Playing Ecosystem.

Game Manager Behaviour

Umpire role

- **Match Start:**
  - GM sends **start** message (game details, role, ..).
  - Players prepare and reply ready.
- **Gameplay Loop:**
  - GM sends **play** message (prev. moves or nil).
  - Players respond with moves in time.
  - GM simulates moves, checks for termination.
- **Game End:**
  - GM sends **stop** message.
  - Computes rewards, stores history.



Game Manager Message Vocabulary

`Message Type`	Explanation
`info`	An info message is used to confirm that a player is up and running.
`start(Id, R, D, S, N)`	A start message to a player for initializing a game instance with Id, stating the player's role R, the game description D, and asking for a move in S seconds from now (N).
`play(Id, M)`	A play message is used to request a move from a player. It includes an identifier Id for the match and a record of the moves M of all players on the preceding step.
`stop(Id, M)`	A stop message is used to tell a player that a match has reached completion (M as before).
`abort(Id)`	An abort message is used to tell a player that match Id is terminating abnormally. It differs from a stop message in that the match need not be in a terminal state.

Starting a Game: Message Exchange

Game Manager (GM) initiates a match with `start` messages to each player.

- Each player replies once initialized for the match.

```
GM → row:  start(m23, row, [role(row), role(col), ...], 10, 10)
GM → col:  start(m23, col, [role(row), role(col), ...], 10, 10)

row → GM:  ready
col → GM:  ready

GM → row:  play(m23, row, nil)
GM → col:  play(m23, col, nil)

row → GM:  coop
col → GM:  defect

GM → row:  stop(m23, [coop,defect])
GM → col:  stop(m23, [coop, defect])

row → GM:  done
col → GM:  done
```

Observations on GDL and Current Encoding:

- **Syntax:** GDL 1.0 uses prefix (Lisp-style) syntax, which can be unintuitive and verbose for rule writing.
- **Assumptions:** Originally for turn-based, perfect-information games.
- **Mismatch:** Our PD scenario involves simultaneous moves and hidden information — not a natural fit for GDL 1.0.
- **Workaround:** Turn-taking is simulated using `control/1` and `noop` actions; true support requires GDL-II (discussed later).

Modelling Questions:

- Why define roles as `role(row)` instead of alternatives like:
  - `player(p1).`
  - `role(p1, row).`
- `input/2` vs `possible/2`?

Interpreting GDL as a Normal Logic Program

GDL based on Normal Logic Programs

Motivation

- While extensions of **GDL 1.0** exist (see later), they are often:
  - Not publicly available, or
  - Difficult to access, modify, or extend
- Our approach: develop a **logic programming solver in Prolog** that:
  - Simulates the `Datalog`-style reasoning of GDL 1.0
  - Provides a more manageable and extensible foundation
  - Enables rapid experimentation with new extensions
- **Idea**: enable easier prototyping of GDL extensions in a declarative environment.

Revised Game Vocabulary

`Core Vocabulary` [8]	**Explanation**
`player(P)`	P is a player in the game.
`role(P, R)`	R is the role of player P in the game.
`initial(I)`	Declares the initial state I of the game.
`initially(F, I)`	Fact F holds in the initial state I of the game.
`move(P, A)`	A move linking the action A taken by a player P.
`possible(M, S)`	M is a possible move in the current state S of the game.
`legal(M, S)`	Move M is legal in state S.
`effect(F, M, S)`	Fact F is true in the next state, if move M is performed in state S.
`holds(F, S)`	Fact F is true in the current state S.
`final(S)`	Defines when S is the final state.
`finally(F, S)`	Defines the facts F holding in the final S.
`goal(P, U)`	Player P gets payoff U in the current state.
`game(I, F)`	The legal evolutions of a game from some state I to a state F.

[8] As before, we use distinct(X,Y) stating $X \neq Y$.

Solver: Domain-Independent Part[9]

Game Execution 'Semantics'

Top-level of our solver:

```
game(F, F) :-
    final(F).

game(S, F) :-
    not final(S),
    legal(M, S),
    game(do(M, S), F).
```

States and Moves:

Abstract transition function `do(M, S)`.

Observations:

- Defines all legal **execution traces** from a starting state `S` to a final state `F`.
- Can be used as a **generator** to enumerate possible plays from the initial state.
- Can be used as a **tester** to verify legality of a concrete final state.

[9] A. Mensfelt, K. Stathis, and V. Trencsenyi, "Autoformalization of game descriptions using large language models," *First International Workshop on Next-Generation Language Models for Knowledge Representation and Reasoning*, 2024.

Solver: Domain-Independent Part

Reasoning about game transitions in the Situation Calculus[10]

Key Design Decisions:

- Must define S and M:
  - State labelled as:

    $\underline{s_0}$, $\underline{do(m_1, s_0)}$, $\underline{do(m_2, do(m_1, s_0))}$, ...
  - A move M is a term of the form: `move(P, A)`
    `P` is player ID, `A` the action.
- We use the **situation calculus** to reason about state transitions.

Situation calculus for our solver:

```
holds(F, S) :-
    initial(S),
    initially(F, S).

holds(F, do(M, S)) :-
    possible(M, S),
    effect(F, M, S).

holds(F, do(M, S)) :-
    possible(M, S),
    holds(F, S),
    not abnormal(F, M, S).
```

[10] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds., Edinburgh University Press, 1969, pp. 463–502.

Solver: Domain-Independent Part

Reasoning about game transitions in the Situation Calculus[11]

Top-level of our solver:

```
game(F, F) :-
    final(F).

game(S, F) :-
    not final(S),
    legal(M, S),
    game(do(M, S), F).
```

We will see that:

```
    legal(M, S) => possible(M, S).
```

Remove ~~possible/2~~ from holds/2:

```
holds(F, S) :-
    initial(S),
    initially(F, S).

holds(F, do(M, S)) :-
    effect(F, M, S).

holds(F, do(M, S)) :-
    holds(F, S),
    not abnormal(F, M, S).
```

Any S/do(M,S) from game/2 are legal.

[11] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds., Edinburgh University Press, 1969, pp. 463–502.

PD Formalization revisited: Roles and Initial State [12]

Initial Situation:

```
initial(s0).

initially(player(p1), s0).
initially(player(p2), s0).
initially(role(p1, row), s0).
initially(role(p2, col), s0).
initially(control(p1), s0).
initially(control(p2), s0).
```

Explanation:

- Initial situation denoted by `s0`.
- There are two players with unique ids: `p1` and `p2`.
- Each player is assigned a role: `row` or `col`.
- Both players are initially in control - allowed to act.

[12] Our formulation assumes that an agent plays multiple games, hence we add the player/1 and role/2 inside the initial state of the game through initially/2 for reasons of modularity.

PD Formalization revisited: Legal Moves and State Transitions

Legal, Possible, and Effect Rules:

```
legal(move(P, M), S) :-
    possible(move(P, M), S),
    holds(control(P), S).

possible(move(P, defect), S) :-
    holds(player(P), S).
possible(move(P, coop), S) :-
    holds(player(P), S).

effect(did(P, M), move(P, M), S).

abnormal(control(P), move(P, M), S).
```

Explanation:

- A move is `legal` if it is `possible` and player is currently in control.
- It is always possible for players to either cooperate or defect.
- The `effect/3` clause captures how the move alters the state.
- The `abnormal/3` clause indicates that executing a move ends the player's control.

PD Formalization revisited: Payoff Rules (v1)

Hardwiring the payoff matrix in goal/2

```
% --- Both cooperate ---
holds( goal(P1, 3), S):-
    holds(did(P1, coop), S),
    holds(did(P2, coop), S),
    distinct(P1, P2).

% --- Both defect ---
holds( goal(P1, 1), S):-
    holds(did(P1, defect), S),
    holds(did(P2, defect), S),
    distinct(P1, P2).
```

```
% --- Sucker's Payoff ---
holds(goal(P1, 0), S) :-
    holds(did(P1, coop), S),
    holds(did(P2, defect), S),
    distinct(P1, P2).

% --- Temptation Payoff ---
holds(goal(P1, 5), S) :-
    holds(did(P1, defect), S),
    holds(did(P2, coop), S),
    distinct(P1, P2).
```

PD Formalization Revisited: Payoff Rules (v2)

Using the payoff matrix in goal/2 definitions


```
% Row player payoff
holds(goal(P1, U1), S) :-
    holds(role(P1, row), S),
    holds(did(P1, M1), S),
    holds(role(P2, col), S),
    holds(did(P2, M2), S),
    payoff(M1, M2, U1, U2).

% Column player payoff
holds(goal(P2, U2), S) :-
    holds(role(P1, row), S),
    holds(did(P1, M1), S),
    holds(role(P2, col), S),
    holds(did(P2, M2), S),
    payoff(M1, M2, U1, U2).
```

```
% --- Payoff matrix ---
payoff(defect, defect, 1, 1).
payoff(coop, defect, 0, 5).
payoff(defect, coop, 5, 0).
payoff(coop, coop, 3, 3).
```

Notes:

- Payoff matrix externally defined and decoupled from the rules.
- Can extend with game G, e.g.,

  `payoff(G, M1, M2, U1, U2).`

PD Formalization Revisited: Termination Condition


```
% --- Termination Condition ---
final(S) :-
    holds(did(R1, M1), S),
    holds(did(R2, M2), S),
    distinct(R1, R2).

finally(F, S) :- final(S), holds(F, S).
```

GDL-like formalization of the PD in Prolog completed.

- `final/1` plays a similar role to `terminal/0` in GDL, but requires the state of the game.
- `finally/2` is used to test what holds in the final states.

Remarks on our Formalization

- Query below returns eight traces (play scenarios) instead of four:

  ?- game(s0, F).
  F = do(move(p2, defect), do(move(p1, defect), s0)) ;
  F = do(move(p2, coop), do(move(p1, defect), s0)) ;
  F = do(move(p1, defect), do(move(p2, defect), s0)) ;
  ...

- In four of the scenarios, `p1` acts first and `p2` second, while in the other four, `p2` goes first and `p1` follows.
- Why? Because situation term `do(M, S)` allows one move at a time.
- This reflects **interleaved semantics**, not true simultaneity.
- **Simultaneous moves** require a different state representation (more later).

Uses of Our Formalization

Two broad categories of use of our game descriptions:

1. **Game Manager as Umpire**
   1. Initializes, starts, and stops matches.
   2. Ensures that player moves are legal and maintains a consistent shared game state.
2. **Players Using the Game Description**
   1. For strategic reasoning: searching from the initial state toward hypothetical final states with preferred utilities.
   2. For validation: verifying that opponent actions are legal from a given state.

Use of game/2 added value over GDL 1.0.

Example of use in a Player's Strategy

Hypothetical final states

Best move (based on opponent's move in previous game[13])

```
select(P, O, S, M) :-
    not holds(last_move(O, _), S),
    holds(default_move(P, M), S).
select(P, O, S, M) :-
    holds(last_move(O, Mo), S),
    findall(
        Ui-Mi,
        (game(S, F), finally(outcome(P,Mi,Ui,O,Mo,Uo), F), Ui>=Uo),
        Options
    ),
    best_move(Options, M).
```

[13] A. Mensfelt, K. Stathis, and V. Trencsenyi, "GAMA: Generative agents for multi-agent autoformalization," *arXiv preprint arXiv:2412.08805*, 2024.

Implementation Issues (Summary)

Computational Considerations for Using the Situation Calculus

- The formalism remains tractable for games defined over small $n \times n$ payoff matrices, or for game trees with limited depth in extensive-form representations.
- As n grows or the depth of the game increases, computational performance may degrade significantly, requiring practical optimizations such as:
  - **Tabling** of `holds/2` queries to memoize fluent evaluations.
  - **Depth-limiting** `game/2` calls to constrain recursive search.
- For more complex games, the trade-off between representational generality and computational efficiency must be carefully balanced.