Ludax: A GPU-Accelerated Domain Specific Language for Board Games

Graham Todd[◇] **Alexander G. Padula**[♠] **Dennis J.N.J. Soemers**[♣] **Julian Togelius**[◇]

[◇]New York University Tandon
Brooklyn, New York, USA

[♠]ETH Zurich
Zurich, Switzerland

[♣]Maastricht University
Maastricht, the Netherlands

Abstract

Games have long been used as benchmarks and testing environments for research in artificial intelligence. A key step in supporting this research was the development of *game description languages*: frameworks that compile domain-specific code into playable and simulatable game environments, allowing researchers to generalize their algorithms and approaches across multiple games without having to manually implement each one. More recently, progress in reinforcement learning (RL) has been largely driven by advances in *hardware acceleration*. Libraries like JAX allow practitioners to take full advantage of cutting-edge computing hardware, often speeding up training and testing by orders of magnitude. Here, we present a synthesis of these strands of research: a domain-specific language for board games which automatically compiles into hardware-accelerated code. Our framework, `Ludax`, combines the generality of game description languages with the speed of modern parallel processing hardware and is designed to fit neatly into existing deep learning pipelines. We envision `Ludax` as a tool to help accelerate games research generally, from RL to cognitive science, by enabling rapid simulation and providing a flexible representation scheme. We present a detailed breakdown of `Ludax`'s description language and technical notes on the compilation process, along with speed benchmarking and a demonstration of training RL agents. The `Ludax` framework, along with implementations of existing board games, is open-source and freely available.

1 Introduction

For the past 75 years, games have served as vital tests and benchmarks for artificial intelligence research. While many specific games have been completely solved [40] or optimized beyond the abilities of the strongest human players [5, 45], the general space of games remains a fertile ground for measuring improvements in reasoning, planning, and strategic thinking. A critical part of this progress, however, is the ability to test approaches and algorithms on a set of environments that are both diverse and computationally efficient.

To help drive further games and learning research, we introduce `Ludax`: a domain-specific language for board games that compiles into GPU-accelerated code written in the JAX library [3]. `Ludax` draws on two main inspirations: (1) `Ludii` [35], a general purpose description language for board games capable of representing more than 1400 games from throughout history and around the world, and (2) `PGX` [29], a collection of optimized JAX-native implementations of classic board games and video games designed to facilitate rapid training and evaluation of modern reinforcement learning

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Scaling Environments for Agents (SEA).

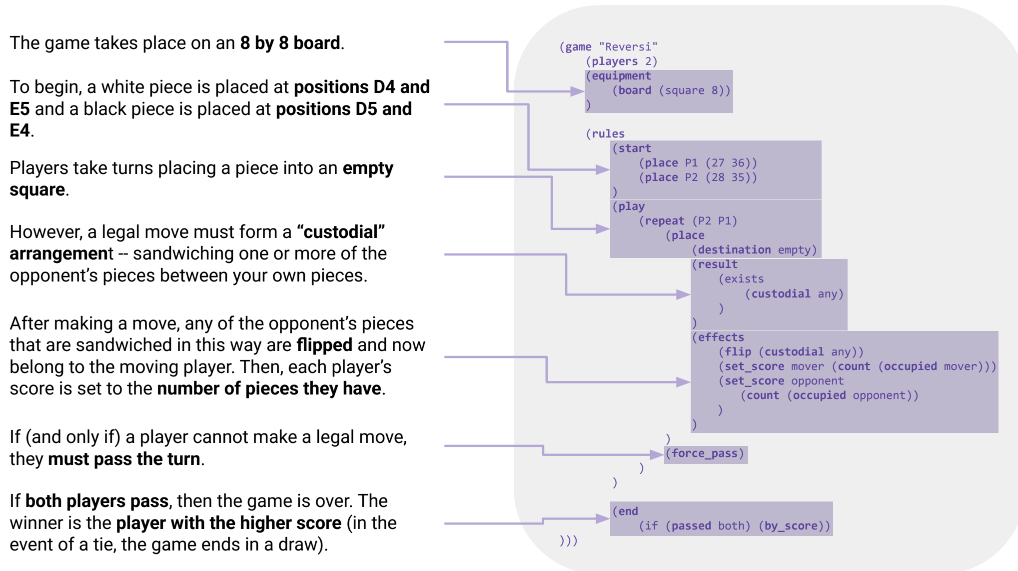

Figure 1: **Natural language description of *Reversi* along with its corresponding translation into** `Ludax`. `Ludax` uses "ludemic" syntax that represents high-level game components as separate program sections.

(RL) agents. `Ludax` presents a flexible and general-purpose game representation format that can be leveraged for efficient simulation and learning on modern computing hardware.

`Ludax` currently supports two-player, perfect-information, turn-based board games played by placing, capturing, and moving pieces. This set of mechanics is broad enough to capture a wide range of existing games (e.g. *Connect Four*, *Pente*, *Hex*, ...) as well as many unexplored *novel* games and variants that fall within that class. Further, `Ludax` is designed to be easily expandable – like with `Ludii`, implementing new game mechanics in `Ludax` only requires implementing new atomic components in the underlying description language. These components can then be combined compositionally with existing elements of the language to produce an entirely new *range* of possible games, instead of each game needing to be implemented separately.

Another design goal for `Ludax` is ease of use, both in terms of game design and experimentation. The syntax of the description language is "ludemic" [35] – splitting game rules into clear sections governing the game's setup, play mechanics, and end conditions. Like with `Ludii`, game programs in `Ludax` resemble English descriptions of rules (see Figure 1). Further, by leveraging the structure of the existing `PGX` library, environments instantiated in `Ludax` can be easily combined with existing frameworks for GPU-accelerated search, reinforcement learning, or evolution [13, 49]. `Ludax` also supports a basic web interface for interactive debugging and potential user-studies.

`Ludax` is fundamentally a platform for accelerating board game research. In an era of increasingly complicated tasks and benchmarks, relatively simple board games may seem to be less interesting research domains (especially as many games have been more-or-less "solved" by modern methods). However, `Ludax` is not just a collection of new tasks. By decoupling rapid execution from the intensive process of writing new environment code, `Ludax` can power new research in a variety of directions. For instance, `Ludax` can be used to analyze RL generalization [46] by defining a wide range of modifications for a target game task (akin to a platform like `Minihack` [39]) or help improve studies of game generation by enabling the rapid evaluation of procedurally-generated rulesets [51, 8]. Finally, `Ludax` can help advance recent research into world modeling [52] by (1) providing a wide and easily-refreshable set of environments to test on efficiently and (2) allowing automated systems to propose and refine world models in these "novel games" by writing high-level and semantically-meaningful DSL code.

To our knowledge, `Ludax` is the first board game description language which compiles into GPU-accelerated code. In the following sections, we provide a detailed description of the language syntax, compilation process, and Ludax's expressive range. We also provide speed benchmarking compared to both Ludii and PGX, as well as an initial demonstration of training learned agents. Finally, we conclude with a discussion of potential use cases and future directions. Ludax is open-source under the Apache 2.0 license, and the code is available at: https://github.com/gdrtodd/ludax.

2 Related Work

Game Description Languages: Game description languages have been used for many years and in a variety of domains. The Stanford GDL [31, 19, 43, 50] is among the most influential, helping to popularize research in general game playing [36] through its use in the International General Game Playing Competition [21, 20]. Other notable examples include VGDL [16, 41, 42] (primarily known from its use in the General Video Game AI framework [34]), RBG [28], Ludi [4], and its successor Ludii [35]. GDLs have also been used to describe the rules of card games [17] as well as to represent human goals in naturalistic simulated environments [11, 12]. Modern game description languages have tended to move away from a basis in formal logic in favor of greater human usability, though there are benefits in efficiency gained by the use of regular languages [27].

GPU-Accelerated Environments: Recent years have seen a proliferation of learning environments implemented in the JAX library or other frameworks that enable hardware (typically GPU) acceleration. Examples include single-agent and multi-agent physics simulators [18, 32, 1], ports of both classic and recent reinforcement learning tasks [10, 30, 29, 33], combinatorial optimization problems [2], multi-agent coordination problems [38], and driving simulators [23, 25]. While these efforts have spurred significant progress and span a wide range of domains and task formulations, each of them implement a fixed environment or set of environments. As such, they cannot easily be extended to novel environments without first writing new hardware-accelerated code. Ludax stands alongside a number of description languages for other domains (e.g. probabilistic programming, planning, single-player puzzles) that leverage JAX for efficient execution [7, 22, 15, 14].

3 Description Language Details

Ludax's game description language draws heavily on the Ludii description language, particularly in its use of "ludemic" syntax that represents game rules in terms of high-level and easily-understandable components [35]. The complete grammar file and syntax details are available in the Supplemental Material.

3.1 Equipment and Start Rules

The `equipment` section contains information about the physical components used by the game. Currently, this only specifies the size and shape of the board (i.e. whether it is square, rectangular, hexagonal, or hexagonal-rectangular). The dimensions and shape of the board are used during compilation to help pre-compute certain game-relevant properties, such as the board indices corresponding to lines of specific lengths. In future versions of Ludax, the equipment section will also detail the different pieces used by each player if the game specifies more than one.

The `start` section is an optional section that contains the rules for the game's setup. For most games, play begins on an empty board and the `start` section is omitted. In some games, such as *Reversi* (see Figure 1), pieces are placed in a particular arrangement at the start of play.

3.2 Play Rules

Typically, the `play` rules of each game are the most involved, as they detail the core mechanics and dynamics of the game. The `play` section is itself broken into one or more subsections called "play phases." Each phase has its own rules for player actions and turn-taking, as well as specific conditions for when to transition to another phase. Most games have only a single phase in which players alternate turns until the game is over, specified with the `repeat` keyword. Some games include a `once_through` phase that progresses through the turn order a single time before advancing to the next phase. The sequence of player turns is specified independently for each phase. For instance, *Yavalax* (Appendix Figure 4, bottom-left panel) begins with the first player making a single move (i.e. `(once_through (P1) ...)`) before both players alternate taking two turns for the rest of the game (i.e. `(repeat (P2 P2 P1 P1) ...)`).

The core of each phase is a "play mechanic" that encodes the ways that players take their turns. In the context of reinforcement learning, a play mechanic specifies both the action space ($\mathcal{A}$) and the transition function ($\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$). At a lower level, each play mechanic also defines a "legal action mask function" that returns whether each action is valid from the current game state. Currently, `Ludax` supports only one kind of play mechanic: `place`. A `place` mechanic's primary argument is a `destination` constraint which specifies where a piece may be placed on a given player's turn. For many games, such as *Tic-Tac-Toe*, this is simply the set of empty board positions. For some games, however, the destination constraint is more involved: in *Connect Four* (Appendix Figure 4, top-right panel), legal actions are empty spaces that are on the bottom edge of the board or immediately above an occupied position (see subsection 3.4 for a discussion of how actions are represented more generally in `Ludax`). Even further, some games have what we call `result` constraints which require that a legal action results or doesn't result in a board with specific properties. *Yavalax* and *Reversi* both use `result` constraints: the former forbids players from placing a piece that forms a line of five or that forms only a single line of four, whereas the latter requires players to place a piece in a way that "sandwiches" one or more of their opponent's pieces in a line. Finally, a play mechanic may optionally specify one or more `effects` that modify the game state after the action is performed. Effects are used to handle mechanics like capturing or flipping pieces, as well as updating each player's score (if the game uses score). Both *Reversi* and *Pente* (Appendix Figure 4, bottom-right panel) use play effects to handle flipping and capturing pieces, respectively, with *Pente* also using the score as an alternate winning condition.

Throughout this section, we have been referring to various properties of a game state and relationships between pieces / positions (e.g. whether pieces are "sandwiched," whether a line is formed, whether a piece is adjacent to another, ...). These are the lowest-level component's of `Ludax`'s description language and are referred to collectively as `masks`, `functions`, and `predicates`. A `mask` takes in the current game state and returns a boolean array over each position on the board. Some masks, like `occupied` or `edge`, take additional grammatical arguments which might specify a particular player or region of the board.[1] A `function` similarly takes in the current game state and returns a single non-negative integer. In `Ludax`'s current form, `line` is probably the most commonly-used `function` – it returns the number of contiguous lines of a given player's pieces on the board, with a specified length and orientation. Lastly, a `predicate` maps from a game state to a single boolean truth value. Many predicates operate over the outputs of `masks` and `functions`, such as `exists` or `equals`, though some like `mover_is` are computed directly from game states. Crucially, the outputs of `masks`, `functions`, and `predicates` can be combined compositionally using first-order logic (excluding quantification) to form more complicated expressions. So, the condition "*if Player 2 makes a line of 4 in a row or a diagonal line of 3...*" would be rendered as follows:

```
(and (mover_is P2) (or (line 4) (line 3 orientation:diagonal)))
```

Note that, for ease of use, `Ludax` automatically interprets the presence of a bare `function` inside a boolean operator as indicating a non-zero value. So, `(line 4)` is equivalent to `(>= (line 4) 1)`.

3.3 End Rules

The last section of a game description in `Ludax` details the criteria that terminate a game. The `end` section contains one or more "end conditions" – these are applied *in order*, with the first condition to activate determining the ending behavior (i.e. which player wins or if the game ends in a draw). If none of the conditions activate, then the game continues. For instance, *Tic-Tac-Toe* includes both the end conditions `(if (line 3) (mover win))` and `(if (full_board) (draw))`, with the draw condition only triggering if the "three in a row condition" is not met. End conditions also frequently refer to a player's score, which is updated or set as a result of an action's effects (see above).

[1] The `adjacent` mask is a special case – it takes *another* mask as an additional argument and returns the board positions adjacent to any of the active positions in the original mask.

3.4 Design Considerations

While Ludax draws heavily from the Ludii description language, there are some important differences which go beyond just changes in syntax. The first of these relates to how both systems represent a game's action space. One of the design goals of Ludii is that game descriptions should resemble as much as possible the rules in natural language. In *Connect Four*, for instance, players take a move by dropping a piece into one of seven columns of the board, at which point the piece falls until it reaches the bottom or rests on another piece. Accordingly, the canonical representation of *Connect Four* in Ludii features pieces that "Drop" into the "LastColumn" chosen by the player (PGX implicitly represents the game in a similar way). As mentioned above, however, Ludax represents the action space differently: players simply place a piece onto an empty board cell, with actions that are not directly above an existing piece or the bottom of the board marked as illegal. Mechanically, the two implementations of *Connect Four* are identical – the difference lies in how they are encoded (especially to simulated players or reinforcement learning agents). The "column-based" representation has many advantages (it matches the physical properties of the game in real life and lowers the branching factor), but it is also *game-specific*. While Ludax also strives to represent game descriptions intuitively, we primarily aim to provide a unified representation format across games, such that general game-playing agents can more easily transfer knowledge and expertise from one game to another. As such, the size and form of the action space for any place-based game is determined only by the size and shape of the board. This choice is also partially motivated by the specifics of working with the JAX library (see Section 4) and has implications for benchmarking and downstream use-cases (see Section 6).

4 Compiling Game Descriptions into Game Environments

In this section, we describe the high-level approach used to map from programs in the Ludax game description language to hardware-accelerated simulation environments. While Ludax specifically instantiates board game environments using the Lark Python library, the general approach is flexible enough to be used with different domains and parsing toolkits. Broadly speaking, Ludax operates by defining the leaves of the grammatical parse tree (i.e. individual masks, functions, and predicates) as atomic functions written in JAX, which are then dynamically composed from the bottom-up to form higher-level operators used by the environment class. Consider again the following game expression:

```
(and (mover_is P2) (or (line 4) (line 3 orientation:diagonal)))
```

During compilation, the leaf-level nodes (i.e. (mover_is P2) and (line 4)) are converted into JAX functions which map from the current game state to (in this case) a boolean truth value, and those functions are then passed up the parse tree. Higher-level nodes, such as (and ...), receive the JAX functions corresponding to each of their children and return a *new* JAX function that also takes the game state as input and implements the appropriate operation (in this case, boolean conjunction). In pseudocode, using the Lark library's Transformer paradigm, this looks like the following:

```
def predicate_and(self, children):
    def predicate_fn(state):
        children_values = [child_fn(state) for child_fn in children]
        return all(children_values)

    return predicate_fn
```

In actuality, both the "children functions" and the combined "predicate function" must be written to be compatible with JAX's vectorization scheme and just-in-time (JIT) compilation. This imposes a number of implementational constraints, most notably that the size and shape of all arrays must be fixed at compile time. This means, for instance, that the dimensions of the "legal action mask" (and, hence, the size of the action space in general) cannot change as the game progresses. In addition, values like the number of iterations in a loop or the positions of a lookup mask must essentially be "pre-specified." Crucially, however, values that are determined during *parsing* (such as the number of children for a given node, or the value of any arguments) can be safely passed into compiled JAX functions as static constants. This fact is what allows Ludax to create JAX functions *dynamically* that nonetheless obey the constraints of vectorization and JIT compilation. At the top of the parse tree, these composed JAX functions are ultimately used to define the behaviors that appear in the environment's `step` function, such as applying the player's action to the board and handling move effects.

We next discuss some of the specific optimizations used by `Ludax`. In general, these are not *global* optimizations: they apply only to certain compositions of game rules and mechanics. Our approach is to deploy these optimizations when they are available and to "fall back" on slower but more general solutions when they are not.

Precomputation: An important optimization used by the `PGX` library (and JAX environments more generally) is to express functions as batched matrix operations rather than iterative procedures. For instance, rather than checking for a line of pieces in *Tic-Tac-Toe* by starting at the position of the last move and scanning out in each direction (as `Ludii`'s implementation does), `PGX` hard-codes the set of board indices that correspond to each possible line of three in the game (i.e. `[[0, 1, 2], [0, 3, 6], [0, 4, 7], ...]`) and performs a single multi-dimensional index into the board array – if any of the of the board index triples all correspond to positions occupied by a single player, then the game is over. `Ludax` adopts and generalizes this approach: during parsing of `line`, for example, the line indices are computed with respect to the size and shape of the game board (i.e. rectangular, hexagonal, ...) as well as the length and orientation of the desired line (i.e. diagonal, vertical, ...). Again, because these values depend only on attributes that are determined during parsing, they can be passed into JAX functions as constants. Precomputation naturally causes a trade-off between compile-time and run-time efficiency. In our case, we opt to use precomputation whenever possible, though some `masks` and `functions` cannot be expressed this way.

Dynamic State Attributes: Different games require tracking different kinds of information about the current game state. Most obviously, some games track a score for each player while others do not. When `Ludax` compiles a game, it automatically extracts the attributes required to instantiate a game state and omits the others, thereby reducing the memory footprint of the entire state object. More importantly, `Ludax` also automatically adds intermediary computations to each call of the environment's `step` function that help speed up later `mask`, `function`, or `predicate` evaluations. For example, in *Hex*, the game ends when one player manages to connect two opposite sides of the board with a continuous path of their pieces. Naively, checking whether the edges of the board are linked requires the expensive step of computing the board's connected components after each move. However, *updating* the board's connected components as a result of placing a single piece can be done very efficiently (a technique used well in the `PGX` implementation). At compile time, `Ludax` determines whether a game makes use of a "connection" rule and modifies the `step` function to iteratively update and track the board's connected components if so, greatly speeding up later checks. In future extensions, this functionality will be used to accommodate games with atypical or computationally expensive rules without affecting the runtime of existing games.

5 Expressive Range

As mentioned above, `Ludax` currently supports a relatively narrow class of games: two-player, perfect-information board games played by placing, capturing, and sliding a single kind of game piece. Despite this, `Ludax`'s description language remains quite expressive. In addition to simple $m - n - k$ line completion games, `Ludax` supports complex win conditions (e.g. *misère* variants, score-based victory), asymmetric player goals, piece capturing and flipping, directional adjacency checks and restrictions, "custodial" mechanics, and games based on connecting arbitrary board regions. `Ludax` also supports regular rectangular and hexagonal boards of arbitrary sizes, as well as "hexagonal-rectangular" boards (e.g. as used in *Hex*). These components can then be combined compositionally to form a wide array of unique mechanics and dynamics. In addition, because `Ludax` is a general description language, implementing a single new game component expands the entire *space* of games in the framework. While the class of games representable in `Ludax` may at present be smaller than that of `Ludii` or other game description languages, it remains expansive. For example, `Ludax` is able to encode both *Yavalath* and *HopThrough* (see Figure 2) – games produced by the `Ludi` [4] and `GAVEL` [51] systems respectively, which were designed to automatically search through the space of games for novel exemplars. In Appendix E we detail a preliminary experiment on automatic game generation in `Ludax` via language models, where we find that two state-of-the-art open-weight LLMs were able to generate novel and potentially interesting games in `Ludax` without any finetuning

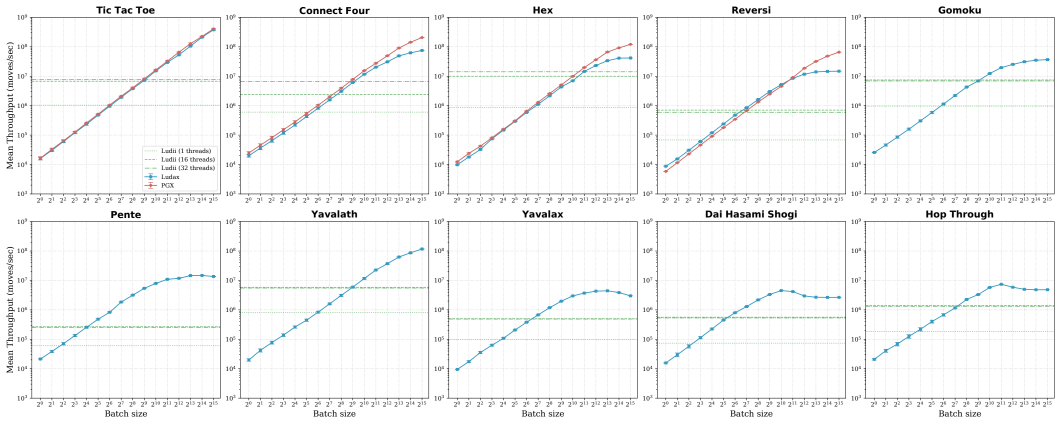

Figure 2: **Average throughput (moves per second) on various exemplar games for** `Ludax`**,** `Ludii`**, and** `PGX`**.** The first four games are implemented in all three frameworks, while the remaining games are implemented only in `Ludax` and `Ludii`. Speeds for `Ludax` and `PGX` are reported for 500 episodes of various batch sizes on a workstation with a single NVIDIA 4090 GPU and 36 threads on 18 CPU cores, while speeds for `Ludii` are reported for parallel execution on the same workstation across 1, 16, and 32 threads. Error bars are standard deviations over the 500 episodes.

or evolutionary search. This provides exciting initial evidence that `Ludax` is well-suited for game generation, and in future work it could serve as a meta environment for training general game-playing RL agents across the entire domain of expressible games.

6 Benchmarking

We benchmark the speed of `Ludax` on a set of 10 games, 4 of which are also implemented in both `Ludii` and `PGX` (allowing for a full comparison) and 6 of which are implemented only in `Ludax` and `Ludii`. Again, we emphasize that these 10 games are just *exemplars* of the class of games which `Ludax` supports, not an exhaustive list. A full description of each benchmark game is available in the Supplementary Material. We perform each of our benchmarking experiments on a workstation with a single NVIDIA 4090 GPU and an Intel i9-9980XE processor (36 threads on 18 CPU cores and 128GB of memory). In Figure 2 we plot the throughput (in steps per second) under a uniformly random action policy for each game environment against the batch size (log scale on both axes), with the standard deviation of throughputs across episodes as error bars. `Ludii` supports parallelization via multi-threading: we report throughput on the same workstation when parallelized on 1, 16, and 32 threads. Evaluations for `Ludax` and `PGX` were obtained by performing 100 warmup full-game episodes at the specified batch size, followed by measuring the speed over 500 episodes, with each evaluation taking at most a few minutes to complete. Evaluations for `Ludii` were obtained by running warmup episodes for 10 seconds, followed by measuring the speed over 30 seconds of episodes.[2] For games with potentially unbounded length (e.g. *Dai Hasami Shogi*), we terminate games for both `Ludax` and `Ludii` after 200 total turns.

Overall, `Ludax` achieves speeds that are competitive with state-of-the-art JAX environments. At small batch sizes, its throughput is similar to that of the `PGX` implementations. At larger batch sizes in more complicated games (i.e. *Hex* and *Reversi*), `PGX` takes a clear edge – though `Ludax` remains within an order of magnitude of `PGX`. The comparative "plateauing" of `Ludax`'s speed at high batch sizes may be due to memory pressure – for instance, `Ludax`'s implementation of *Hex* maintains both a board and the connected components for each game state, whereas the `PGX` implementation cleverly combines both into a single array. This kind of optimization is of course theoretically implementable in `Ludax` as well, though again we emphasize the desiderata of avoiding *game-specific* solutions.

`Ludax` also outspeeds `Ludii` on 16 and 32 threads across all 10 games, achieving a maximum speedup of between ~3x (*Hex*) and ~55x (*Pente*). We note that there are factors that both advantage

[2]We opted to measure speed for `Ludax` and `PGX` using a fixed number of episodes because JAX's compilation procedure makes it difficult to halt execution after a specific elapsed wall time.

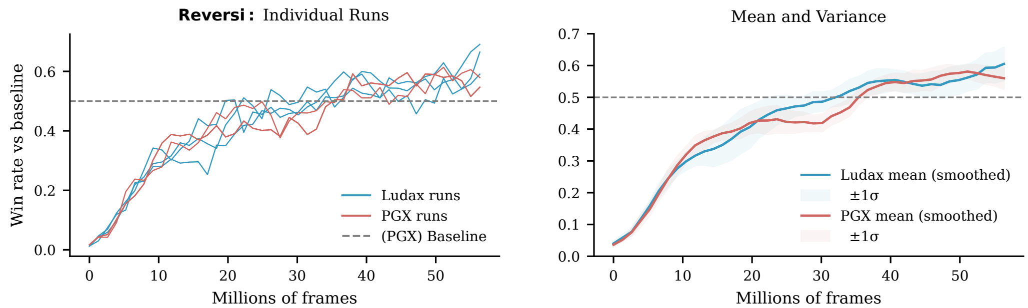

Figure 3: **Performance of reinforcement learning agents trained in the `Ludax` and `PGX` implementations of *Reversi* against the `PGX` baseline agent.** On the left, we plot the average winrate of the learned agents against the baseline over time and across three separate runs. On the right, we plot the average and variance of the winrates. Each run took roughly 3 hours to complete on a workstation with a single A100 GPU.

and disadvantage `Ludax` in this specific comparison against `Ludii`. One potential advantage for `Ludax` is its smaller representation space – implementations of basic mechanics in `Ludii` support a wider range of optional arguments and board types, with a corresponding increase in computational overhead (though see Section 4 for how this may be avoided). Conversely, `Ludii`'s ability to use dynamically-sized data structures brings advantages that are particularly beneficial in uniformly random playouts, but would (partially) disappear in playouts using deep reinforcement learning. `Ludii` also has optimized playout implementations tailored towards many of the categories of games covered by `Ludax` [47], though these optimizations are also more difficult to apply in the context of deep learning.

7 Learned Agents

Finally, we demonstrate the feasibility of training reinforcement learning agents using the `Ludax` framework. We train our agent on the game *Reversi* (also known as *Othello*) using the `AlphaZero`-style [45] training script from the `PGX` library[3] (making only slight modifications to accommodate minor differences between the `Ludax` and `PGX` APIs). We use the same ResNetV2 [24] network architecture and training hyperparameters as `PGX` (full details available in the Supplementary Material) and train three separate runs on a single A100 GPU. Each run lasted roughly 57 million frames and took roughly three hours to complete.

We compare the performance of agents trained in the `Ludax` and `PGX` environments against the baseline *Reversi* agent provided by the `PGX` library in Figure 3. Evaluations were performed by playing two batches of 1024 games (one with the learned agent as the first player and one as the second player), with actions sampled from the normalized output of the policy head at each step. We see that both learned agents achieve remarkably similar performances against the baseline, with little to no differences in learning speed or stability. While a more thorough, tournament-based evaluation would be necessary to properly rank the agents against each other, our objective is to demonstrate the general success of the training procedure and not to definitively defeat the baseline agent. Although the `PGX` implementation of the *Reversi* environment is slightly more efficient, this translated into only marginal improvements in overall runtime (about 1.5%) owing to the shared overhead of network forward passes and weight updates. Like `PGX`, `Ludax` offers a familiar API and an efficient set of implementations with which to train learned player agents.

8 Limitations

Generality: As mentioned in Section 5, `Ludax` currently supports a smaller class of games than other comparable game description languages. While we aim to increase the range of games expressible in `Ludax` (see below), it will likely never match the full generality of `Ludii`. As such, other frameworks

[3] https://github.com/sotetsuk/pgx/blob/main/examples/alphazero/train.py (used under Apache 2.0 license)

may be more appropriate for use-cases in which a broad range of games is more important than rapid simulation. Further, Ludax does not support genres other than board games (e.g. video games, card games, ...) – we leave the development of hardware accelerated description languages for such domains as an exciting area of future work.

Efficiency: Compared to bespoke JAX implementations of board games (such as in the PGX library), environments in Ludax have slightly worse throughput – though the gap is marginal in a standard RL training setup. We deploy a number of optimizations to help close the efficiency gap when possible (see Section 4), but there are ultimately unavoidable trade-offs between speed and generality. For the purpose of training or benchmarking single-task agents on existing games, hard-coded simulators may remain the superior choice.

9 Future Work

The most obvious avenue of extension for Ludax is the implementation of additional game mechanics. In particular, we aim to support irregular board shapes, games with multiple piece types (e.g. *Checkers*) and games with multiple distinct gameplay phases (e.g. *Nine-Men's Morris*). In addition, it's also very likely that the implementation of specific gameplay elements could be further optimized for throughput and / or memory footprint. However, a balance must be struck between efficiency and generality: a less efficient solution which accommodates all valid games under the grammar is ultimately preferable to one which only applies to a subset of games. Lastly, we aim to provide a more robust visual interface for Ludax, both for the purpose of facilitating human-subject research (e.g. with packages like NiceWebRL[6]) and the potential development of more "human-like" artificial agents which process the game board at the pixel level and select actions spatially.

We are particularly excited about the potential application of Ludax to the study of *automated game design* (or reward-guided program synthesis more generally [9, 48, 37]). Such systems depend on both a broad representation space and rapid evaluation of novel games – see Appendix E for a preliminary investigation of Ludax's suitability for such research. The efficiency of Ludax may also make it possible to train a reinforcement learning agent from scratch as part of the inner loop of game evaluation, potentially unlocking a new range of computational features (e.g. learning curves) that correlate with human notions of fun and engagement. Relatedly, Ludax may prove useful to research on *human behavior and play*. Recent work has explored heuristic-based computational models of human play on simple line completion games [53], and Ludax offers the possibility to both accelerate computation and broaden the domain to a wider class of games. Finally, Ludax offers an avenue to extend recent research in *general game playing* (e.g. with large language models [44]) by providing a wide base of efficient game implementations that can in turn be leveraged for tree search algorithms or training world models.

10 Conclusions

We introduce a novel framework for games research that combines the generality of game description languages with the efficiency of modern hardware-accelerated learning environments. Our framework, Ludax, represents a broad class of two-player board games and compiles directly into code in the JAX Python library. Games in Ludax achieve speeds that are competitive with hand-crafted JAX implementations and faster than the widely-used Ludii game description language, and Ludax environments can easily be deployed in existing pipelines for deep reinforcement learning. Our framework helps widen and accelerate games research, with the potential to unlock new approaches in RL generalization, automatic game generation, and cognitive modeling.

Broader Impact

This paper presents a general framework with the goal of advancing reinforcement learning and games research. While there are many potential societal consequences of such work in general, we do not feel that any must be specifically highlighted here. Ludax does not use or reproduce any copyrightable game material (i.e. art, specific expressions of rules, or game code).

Acknowledgements

The authors would like to thank Sam Earle for many fruitful discussions and valuable insights on JAX programming and language design. Author GT acknowledges support from the National Science Foundation Graduate Research Fellowship under Grant DGE-2234660.

References

[1] M. Bettini, R. Kortvelesy, J. Blumenkamp, and A. Prorok. VMAS: A vectorized multi-agent simulator for collective robot learning. In *Proceedings of the 16th International Symposium on Distributed Autonomous Robotic Systems*, DARS '22. Springer, 2022.

[2] C. Bonnet, D. Luo, D. Byrne, S. Surana, V. Coyette, P. Duckworth, L. I. Midgley, T. Kalloniatis, S. Abramowitz, C. N. Waters, et al. Jumanji: a diverse suite of scalable reinforcement learning environments in jax. *CoRR*, 2023.

[3] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[4] C. B. Browne. *Automatic Generation and Evaluation of Recombination Games*. Phd thesis, Faculty of Information Technology, Queensland University of Technology, Queensland, Australia, 2009.

[5] M. Campbell, A. J. Hoane Jr, and F.-h. Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[6] W. Carvalho, V. Goddla, I. Sinha, H. Shin, and K. Jha. Nicewebrl: a python library for human subject experiments with reinforcement learning environments. *arXiv preprint arXiv:2508.15693*, 2025.

[7] K. Chandra, T. Chen, J. B. Tenenbaum, and J. Ragan-Kelley. A domain-specific probabilistic programming language for reasoning about reasoning (or: a memo on memo). 2025.

[8] K. M. Collins, G. Todd, C. E. Zhang, A. Weller, J. Togelius, J. Chu, L. Wong, T. Griffiths, and J. B. Tenenbaum. Generation and evaluation in the human invention process through the lens of game design. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 47, 2025.

[9] C. Cui, W. Wang, M. Zhang, G. Chen, Z. Luo, and B. C. Ooi. Alphaevolve: A learning framework to discover novel alphas in quantitative investment. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 2208–2216. ACM, June 2021.

[10] S. Dalton and I. Frosio. Accelerating reinforcement learning through gpu atari emulation. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 19773–19782. Curran Associates, Inc., 2020.

[11] G. Davidson, T. M. Gureckis, and B. Lake. Creativity, compositionality, and common sense in human goal generation. In *Proceedings of the annual meeting of the cognitive science society*, volume 44, 2022.

[12] G. Davidson, G. Todd, J. Togelius, T. M. Gureckis, and B. M. Lake. Goals as reward-producing programs. *Nature Machine Intelligence*, 7(2):205–220, 2025.

[13] DeepMind, I. Babuschkin, K. Baumli, A. Bell, S. Bhupatiraju, J. Bruce, P. Buchlovsky, D. Budden, T. Cai, A. Clark, I. Danihelka, A. Dedieu, C. Fantacci, J. Godwin, C. Jones, R. Hemsley, T. Hennigan, M. Hessel, S. Hou, S. Kapturowski, T. Keck, I. Kemaev, M. King, M. Kunesch, L. Martens, H. Merzic, V. Mikulik, T. Norman, G. Papamakarios, J. Quan, R. Ring, F. Ruiz, A. Sanchez, L. Sartran, R. Schneider, E. Sezener, S. Spencer, S. Srinivasan, M. Stanojević, W. Stokowiec, L. Wang, G. Zhou, and F. Viola. The DeepMind JAX Ecosystem, 2020.

[14] S. Earle, G. Todd, Y. Li, A. Khalifa, M. U. Nasir, Z. Jiang, A. Banburski-Fahey, and J. Togelius. Puzzlejax: A benchmark for reasoning and learning, 2025.

[15] S. Earle and J. Togelius. Autoverse: Evolving symbolic neural cellular automata environments to train player agents. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 175–178, 2025.

[16] M. Ebner, J. Levine, S. M. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. 2013.

[17] J. M. Font, T. Mahlmann, D. Manrique, and J. Togelius. A card game description language. In *Applications of Evolutionary Computation: 16th European Conference, EvoApplications 2013, Vienna, Austria, April 3-5, 2013. Proceedings 16*, pages 254–263. Springer, 2013.

[18] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem. Brax-a differentiable physics engine for large scale rigid body simulation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

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

[20] M. R. Genesereth and Y. Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–111, 2013.

[21] M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[22] M. Gimelfarb, A. Taitler, and S. Sanner. Jaxplan and gurobiplan: Optimization baselines for replanning in discrete and mixed discrete and continuous probabilistic domains. In *34th International Conference on Automated Planning and Scheduling*, 2024.

[23] C. Gulino, J. Fu, W. Luo, G. Tucker, E. Bronstein, Y. Lu, J. Harb, X. Pan, Y. Wang, X. Chen, J. D. Co-Reyes, R. Agarwal, R. Roelofs, Y. Lu, N. Montali, P. Mougin, Z. Yang, W. B, A. Faust, R. McAllister, D. Anguelov, and B. Sapp. Waymax: An accelerated, data-driven simulator for large-scale autonomous driving research. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2023.

[24] K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision – ECCV 2016*, pages 630–645, Cham, 2016. Springer International Publishing.

[25] S. Kazemkhani, A. Pandya, D. Cornelisse, B. Shacklett, and E. Vinitsky. Gpudrive: Data-driven, multi-agent driving simulation at 1 million fps. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025.

[26] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, Berlin, Heidelberg, 2006.

[27] J. Kowalksi, R. Miernik, M. Mika, W. Pawlik, J. Sutowicz, M. Szykuła, and A. Tkaczyk. Efficient reasoning in regular boardgames. In *Proceedings of the 2020 IEEE Conference on Games*, pages 455–462. IEEE, 2020.

[28] J. Kowalski, M. Maksymilian, J. Sutowicz, and M. Szykuła. Regular boardgames. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, volume 33, pages 1699–1706. AAAI Press, 2019.

[29] S. Koyamada, S. Okano, S. Nishimori, Y. Murata, K. Habara, H. Kita, and S. Ishii. Pgx: Hardware-accelerated parallel game simulators for reinforcement learning. *Advances in Neural Information Processing Systems*, 36:45716–45743, 2023.

[30] R. T. Lange. gymnax: A JAX-based reinforcement learning environment library, 2022.

[31] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, 2008.

[32] V. Makoviychuk, L. Wawrzyniak, Y. Guo, M. Lu, K. Storey, M. Macklin, D. Hoeller, N. Rudin, A. Allshire, A. Handa, and G. State. Isaac gym: High performance GPU based physics simulation for robot learning. In J. Vanschoren and S. Yeung, editors, *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran, 2021.

[33] M. Matthews, M. Beukman, B. Ellis, M. Samvelyan, M. Jackson, S. Coward, and J. Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2024.

[34] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas. General video game AI: A multitrack framework for evaluating agents, games, and content generation algorithms. *IEEE Transactions on Games*, 11(3):195–214, 2019.

[35] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – the ludemic general game system. In G. D. Giacomo, A. Catala, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, editors, *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418. IOS Press, 2020.

[36] J. Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.

[37] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[38] A. Rutherford, B. Ellis, M. Gallici, J. Cook, A. Lupu, G. Ingvarsson, T. Willi, A. Khan, C. S. de Witt, A. Souly, et al. Jaxmarl: Multi-agent rl environments in jax. *CoRR*, 2023.

[39] M. Samvelyan, R. Kirk, V. Kurin, J. Parker-Holder, M. Jiang, E. Hambro, F. Petroni, H. Küttler, E. Grefenstette, and T. Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Advances in Neural Information Processing Systems*, 2021.

[40] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Checkers is solved. *Science*, 317(5844):1518–1522, 2007.

[41] T. Schaul. A video game description language for model-based or interactive learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, pages 193–200. IEEE, 2013.

[42] T. Schaul. An extensible description language for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331, Dec. 2014.

[43] S. Schiffel and M. Thielscher. Representing and reasoning about the rules of general games with imperfect information. *Journal of Artificial Intelligence Research*, 49:171–206, 2014.

[44] J. Schultz, J. Adamek, M. Jusup, M. Lanctot, M. Kaisers, S. Perrin, D. Hennes, J. Shar, C. Lewis, A. Ruoss, et al. Mastering board games by external and internal planning with language models. *arXiv preprint arXiv:2412.12119*, 2024.

[45] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[46] D. J. Soemers, S. Samothrakis, K. Driessens, and M. Winands. Environment descriptions for usability and generalisation in reinforcement learning. In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence*, pages 983–992, 2025.

[47] D. J. N. J. Soemers, É. Piette, M. Stephenson, and C. Browne. Optimised playout implementations for the Ludii general game system. In C. Browne, A. Kishimoto, and J. Schaeffer, editors, *Advances in Computers Games (ACG 2021)*, volume 13262 of *Lecture Notes in Computer Science*, pages 223–234. Springer, Cham, 2022.

[48] A. Surina, A. Mansouri, L. Quaedvlieg, A. Seddas, M. Viazovska, E. Abbe, and C. Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning, 2025.

[49] Y. Tang, Y. Tian, and D. Ha. Evojax: Hardware-accelerated neuroevolution. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 308–311, 2022.

[50] M. Thielscher. GDL-III: A description language for epistemic general game playing. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 1276–1282, 2017.

[51] G. Todd, A. G. Padula, M. Stephenson, É. Piette, D. Soemers, and J. Togelius. Gavel: Generating games via evolution and language models. *Advances in Neural Information Processing Systems*, 37:110723–110745, 2024.

[52] L. Ying, K. M. Collins, P. Sharma, C. Colas, K. I. Zhao, A. Weller, Z. Tavares, P. Isola, S. J. Gershman, J. D. Andreas, et al. Assessing adaptive world models in machines with novel games. *arXiv preprint arXiv:2507.12821*, 2025.

[53] C. E. Zhang, K. M. Collins, L. Wong, A. Weller, and J. Tenenbaum. People use fast, goal-directed simulation to reason about novel games. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 46, 2024.

Appendices

A Example Games and Syntax

Below we present the Ludax syntax for a small set of exemplar games (*Reversi*, *Connect Four*, *Yavalax*, and *Pente*) to help illustrate aspects of Ludax's syntax and structure.

```
(game "Reversi"
    (players 2)
    (equipment
        (board (square 8))
    )

    (rules
        (start
            (place P1 (28 35))
            (place P2 (27 36))
        )
        (play
            (repeat (P1 P2)
                (place
                    (destination empty)
                    (result
                        (exists
                            (custodial any)
                        )
                    )
                    (effects
                        (flip (custodial any))
                        (set_score mover (count (occupied mover)))
                        (set_score opponent
                            (count (occupied opponent))
                        )
                    )
                )
                (force_pass)
            )
        )

        (end
            (if (passed both) (by_score))
)))
```

```
(game "Connect-Four"
    (players 2)
    (equipment
        (board (rectangle 6 7))
    )

    (rules
        (play
            (repeat (P1 P2)
                (place (destination (and
                      empty
                      (or
                            (edge bottom)
                            (adjacent occupied direction:up)
                      )
                )))
            )
        )

        (end
            (if (line 4) (mover win))
            (if (full_board) (draw))
        )
    )
)
```

```
(game "Yavalax"
    (players 2)
    (equipment
        (board (square 13))
    )

    (rules
        (play
            (once_through (P1)
                (place (destination empty))
            )
            (repeat (P2 P2 P1 P1)
                (place
                    (destination empty)
                    (result
                        (and
                            (not (line 5))
                            (not (= (line 4) 1))
                        )
                    )
                )
            )
        )
        (end
            (if (>= (line 4) 2) (mover win))
            (if (full_board) (draw))
        )
    )
)
```

```
(game "Pente"
    (players 2)
    (equipment
        (board (square 19))
    )

    (rules
        (play
            (once_through (P1)
                (place (destination center))
            )
            (repeat (P2 P1)
                (place
                    (destination empty)
                    (effects
                        (capture (custodial 2) increment_score:true)
                    )
                )
            )
        )
        (end
            (if (line 5) (mover win))
            (if (>= (score mover) 10) (mover win))
            (if (full_board) (draw))
        )
    )
)
```

Figure 4: **Ludax syntax for *Reversi* and *Connect Four* (classic board games), as well as *Yavalax* and *Pente* (modern board games).**

B Ludax Grammar

Below we present the complete grammar specification for Ludax, using the syntax of the Lark Python library (raw string constants omitted for brevity).

```
// ---Root---
game: "(game" name players equipment rules rendering? ")"

// ---Players---
players: "(players" positive_int ")"

// ---Equipment---
equipment: "(equipment" board")"
board: "(board" (board_square | board_rectangle | board_hexagon | board
    _hex_rectangle) ")"
board_square: "(square" number ")"
board_rectangle: "(rectangle" number number ")"
board_hexagon: "(hexagon" number ")"
board_hex_rectangle: "(hex_rectangle" number number ")"

// ---Rules---
rules: "(rules" start_rules? play_rules end_rules ")"

// ---Start rules---
start_rules: "(start" start_rule+ ")"
start_rule: start_place
start_place: "(place" player_reference (pattern_arg | multi_mask_arg) ")"

// ---Play rules---
play_rules: "(play" play_phase+ ")"
play_phase: phase_once_through | phase_repeat
phase_once_through: "(once-through" play_mover_order play_super_mechanic ")
    "
phase_repeat: "(repeat" play_mover_order play_super_mechanic ")"
play_mover_order: "(" player_reference+ ")"

play_super_mechanic: play_mechanic force_pass?
play_mechanic: play_place | play_move
force_pass: "(force_pass" ")"

// ---Place rules---
play_place: "(place" mover_reference? place_destination_constraint place
    _result_constraint? play_effects? ")"
place_destination_constraint: "(destination" super_mask ")"
place_result_constraint: "(result" super_predicate ")"

// ---Move rules---
play_move: "(move" move_types move_source_constraint
    move_destination_constraint move_result_constraint? play_effects? ")"
move_types: move_type | "(" move_type+ ")"
move_type: move_hop
        | move_slide

move_hop: "hop" | "(hop" direction_arg ")"
move_slide: "slide" | "(slide" direction_arg ")"

move_source_constraint: "(source" super_mask ")"
move_destination_constraint: "(destination" super_mask ")"
move_result_constraint: "(result" super_predicate ")"
```

```
// ---Effects---
play_effects: "(effects" play_effect+ ")"
play_effect: effect_capture
          | effect_flip
          | effect_increment_score
          | effect_set_score

effect_capture: "(capture" super_mask mover_reference? increment_score_arg?
     ")"
effect_flip: "(flip" super_mask mover_reference? ")"
effect_increment_score: "(increment_score" mover_reference function ")"
effect_set_score: "(set_score" mover_reference function ")"

// ---Functions---
function: function_add
       | function_connected
       | function_constant
       | function_count
       | function_line
       | function_multiply
       | function_score
       | function_subtract

function_add: "(add" function+ ")"
function_connected: "(connected" multi_mask_arg mover_reference?
    direction_arg? ")"
function_constant: positive_int
function_count: "(count" super_mask ")"
function_line: "(line" positive_int orientation_arg? exact_arg?
    exclude_arg? ")"
function_multiply: "(multiply" function+ ")"
function_score: "(score" mover_reference ")"
function_subtract: "(subtract" function function ")"

// ---End rules---
end_rules: "(end" end_rule+ ")"
end_rule: "(if" super_predicate end_rule_result ")"
?end_rule_result: result_win | result_lose | result_draw | result_by_score

// -- Result definitions --
result_win: "(" mover_reference "win" ")"
result_lose: "(" mover_reference "lose" ")"
result_draw: "(" "draw" ")"
result_by_score: "(" "by_score" ")"

// -- Mask definitions --
super_mask: mask | super_mask_and | super_mask_or | super_mask_not
super_mask_and: "(and" super_mask+ ")"
super_mask_or: "(or" super_mask+ ")"
super_mask_not: "(not" super_mask ")"

mask: mask_adjacent
    | mask_center
    | mask_column
    | mask_corners
    | mask_corner_custodial
    | mask_custodial
    | mask_edge
```

```
    | mask_empty
    | mask_occupied
    | mask_pattern
    | mask_prev_move
    | mask_row

mask_adjacent: "(adjacent" super_mask direction_arg? ")"
mask_center: "center"
mask_column: "(column" positive_int ")"
mask_corners: "corners"
mask_corner_custodial: "corner_custodial" | "(corner_custodial" mover
    _reference ")"
mask_custodial: "(custodial" custodial_length_arg mover_reference?
    orientation_arg? ")"
mask_edge: "(edge" edge ")"
mask_empty: "empty"
mask_occupied: "occupied" | "(occupied" mover_reference ")"
mask_pattern: "(pattern" dimensions_arg pattern_arg rotate_arg? ")"
mask_prev_move: "(prev_move" mover_reference ")"
mask_row: "(row" positive_int ")"

multi_mask: multi_mask_corners
          | multi_mask_edges
          | multi_mask_edges_no_corners

multi_mask_corners: "corners"
multi_mask_edges: "edges"
multi_mask_edges_no_corners: "edgesNoCorners"

// ---Predicate definitions---
super_predicate: predicate | super_predicate_and | super_predicate_or |
    super_predicate_not
super_predicate_and: "(and" super_predicate+ ")"
super_predicate_or: "(or" super_predicate+ ")"
super_predicate_not: "(not" super_predicate ")"

predicate: predicate_equals
         | predicate_exists
         | predicate_full_board
         | predicate_function
         | predicate_greater_equals
         | predicate_less_equals
         | predicate_mover_is
         | predicate_passed

predicate_equals: "(=" function+ ")"
predicate_exists: "(exists" super_mask ")" // technically equivalent to (>=
     (count mask) 1)
predicate_full_board: "(" "full_board" ")"
predicate_function: function // special syntax which is equivalent to "(>=
    function 1)"
predicate_greater_equals: "(>=" function function ")"
predicate_less_equals: "(<=" function function ")"
predicate_mover_is: "(mover_is" player_reference ")"
predicate_passed: "(passed" (mover_reference | BOTH) ")"

// Additional (potentially optional) arguments for predicates
custodial_length_arg: ANY | positive_int
dimensions_arg: "(" positive_int positive_int ")"
```

```
direction_arg: "direction:" direction
exact_arg: "exact:" boolean
exclude_arg: "exclude:" multi_mask_arg
increment_score_arg: "increment_score:" boolean
multi_mask_arg: multi_mask | super_mask | "(" super_mask+ ")"
orientation_arg: "orientation:" orientation
pattern_arg: "(" positive_int+ ")"
rotate_arg: "rotate:" boolean

// Optional rendering details
rendering: "(rendering" rendering_detail+ ")"
rendering_detail: color_assignment

color_assignment: "(color" player_reference color ")"

// General-purpose definitions
?number: SIGNED_NUMBER
?positive_int: /[0-9]+/
?boolean: TRUE | FALSE
?edge: TOP | BOTTOM | LEFT | RIGHT | TOP_LEFT | TOP_RIGHT | BOTTOM_LEFT |
    BOTTOM_RIGHT
?direction: UP | DOWN | LEFT | RIGHT | UP_LEFT | UP_RIGHT | DOWN_LEFT |
    DOWN_RIGHT | VERTICAL | HORIZONTAL | ORTHOGONAL | DIAGONAL |
    BACK_DIAGONAL | FORWARD_DIAGONAL | ANY
?orientation: VERTICAL | HORIZONTAL | ORTHOGONAL | DIAGONAL | BACK_DIAGONAL
     | FORWARD_DIAGONAL | ANY
?color: WHITE | BLACK
// ----------------------------

?player_reference: P1| P2
?mover_reference: MOVER | OPPONENT
name: STRING
variable_name: /\?[a-z][a-z0-9]*/
id: /[a-zA-Z0-9_]+/
```

C Benchmark Game Descriptions

Below, we present natural language descriptions of the rules for each of the exemplar games analyzed in Section 6.

***Tic-Tac-Toe*:** Players take turns placing a piece into an empty space on a square 3-by-3 board. If a player forms a line of three of their pieces in a row (either vertically, horizontally, or diagonally), they win. If the board is completely full but no lines have been formed, then the game ends in a draw.

***Connect Four*:** Players take turns placing a piece into the top of one of the seven columns on a 6-by-7 board. The piece then "falls" until it rests on either the bottom of the board or another piece. A player can't place a piece into a column that is already "full." If a player forms a line of four of their pieces in a row (either vertically, horizontally, or diagonally), they win. If the board is completely full but no lines have been formed, then the game ends in a draw.

***Hex*:** Players take turns placing a piece into an empty space on an 11-by-11 board composed of hexagonal tiles (forming a parallelogram, see visual depiction here). The objective for the first player is to form a continuous path of their pieces that connects the top edge of the board with the bottom edge, while the objective for the second player is to do the same but connect the left and right edges of the board. The first player to achieve their objective wins the game. Because of the geometric properties of the board, it's not possible for the game to end in a draw.

***Reversi*:** The game takes place on a square 8-by-8 board. To begin, a white piece is placed at positions D4 and E5 and a black piece is placed at positions D5 and E4 (see visual depiction here). Players take turns placing a piece into an empty space such that a line of one or more of the opponent's pieces are "sandwiched" on either end by the player's pieces. This configuration is called a "custodial" arrangement of pieces. After placing a piece, any of the opponent's pieces which are in such a custodial arrangement are flipped and now belong to the player who just moved. It's possible for a single move to form multiple custodial arrangements in different directions, in which case all of the relevant pieces are flipped. If a player cannot make a legal move, they must pass (and they cannot pass without making a move otherwise). If both players pass, then the game is over. The winner is determined by the player who has the largest number of pieces on the board at the end of the game (in the event of a tie, the game ends in a draw).

***Gomoku*:** Players take turns placing a piece into an empty space on a square 15-by-15 board. If a player forms a line of exactly five of their pieces in a row (either vertically, horizontally, or diagonally), they win. However, forming a line of six or more does not count – the player must have at least one line of exactly five. If the board is completely full but no lines of exactly five have been formed, then the game ends in a draw.

***Pente*:** Players take turns placing a piece into an empty space on a square 19-by-19 board. If a player forms a line of five of their pieces in a row (either vertically, horizontally, or diagonally), they win. In addition, if placing a piece causes a line of exactly two of the opponent's pieces to be put into a custodial arrangement, the two pieces are captured and removed from a board. Note that placing a piece *into* a custodial arrangement formed by the opponent does not result in any pieces being captured. A player who captures at least 10 of the opponent's pieces over the course of the game wins. In the variant of *Pente* implemented in `Ludii` and `Ludax`, the first player must make their first move into the exact center of the board.

***Yavalath*:** Players take turns placing a piece into an empty space on a regular hexagonal board with a diameter of 9 spaces. If a player forms a line of four of their pieces in any direction (either diagonally or horizontally[4]), they win. However, if a player forms a line of three of their pieces in a row without also forming a line of four, they lose. If the board is completely full but no lines of four or three have been formed, then the game ends in a draw.

***Yavalax*:** To begin, the first player places a piece into an empty space on a square 13-by-13 board. Starting with Player 2, players then take turns placing two pieces into empty spaces on the board. If a player forms at least two distinct lines of four of their pieces in any direction (either vertically, horizontally, or diagonally), they win. However, a player may not place a piece into a space if doing so would form a line of five pieces in any direction or if it would form exactly one line of four pieces in any direction. Note that this restriction applies to a player's first move of their turn even if they could form a second line of four pieces with their second move of the turn (and thus win). If the board is completely full and neither player has formed at least two distinct lines of four pieces, then the game ends in a draw.

***Dai Hasami Shogi*:** The game takes place on a square 9-by-9 board. To begin, white pieces are placed on the bottom two rows of the board and black pieces are placed on the top two rows. Players take turns moving one of their pieces, either by sliding it any number of squares vertically or horizontally (i.e. as a rook) or by hopping over one piece (belonging to either player) vertically or horizontally into an empty square. Hopping over a piece does not capture it, but opposing pieces can be captured "custodially" (i.e. by moving to surround an enemy piece on both sides vertically or horizontally). An opponent's piece in a corner can also be captured by moving a piece to occupy both orthogonally-adjacent squares. A player wins if they manage to form a horizontal or vertical line of 5 pieces in a row if none of those pieces are in their starting rows.

***HopThrough*:** The game takes place on a square 8-by-8 board. To begin, white pieces are placed on the bottom two rows and black pieces are placed on the top two rows. Players take turns moving one of their pieces by hopping over an adjacent piece (belong to either player) in any direction. Hopping over a piece does not capture it. A player wins if they manage to get one of their pieces to the opposite edge of the board (i.e. the top edge for the first player and the bottom edge for the second player).

[4] `Ludax` assumes a canonical orientation for hexagonal boards in which the diameter stretches from left to right, though it is functionally equivalent to the orientation in which the diameter runs vertically)

```
Invent simple rules for a novel two player abstract strategy game
called {name}. Implement it in the ludax language. You will find
attached the ludax's grammar as well as a few examples of games
implemented in ludax. Start by implementing a simplified version
of your rules, and then incrementally add rules that are harder to
express in ludax. At each step, make sure you write a compilable
game according to ludax's grammar.
```

Listing 1: System instruction for LLM-based generation.

D Training Hyperparameters

Below we provide the exact training hyperparameters used in the reinforcement learning experiments in Section 7. These are largely copied from the PGX implementation.

- **Model architecture:** ResnetV2
- **Number of channels:** 128
- **Number of layers:** 6
- **Self-play batch size:** 1024
- **Self-play simulations:** 32
- **Self-play max steps:** 256
- **Training batch size:** 4096
- **Learning rate:** 0.001
- **Evaluation frequency:** 5
- **Training iterations:** 219

Note that each "iteration" consists of generating play data for 256 steps using the self-play batch size of 1024 (see [29]). We train the model for 219 iterations, which corresponds to $256 \times 1024 \times 219 = 57409536$ (or roughly 57 million) steps in the environment.

E Game Generation

We attempt to synthesize new games in the Ludax DSL using two approaches: random sampling and LLM-based generation. In Table 1, we present the GAVEL game evaluation metrics for each method.

Random Sampling: Games are generated by naive uniform random sampling. Starting from the root game "ludeme" (i.e. production rule), we sample the next ludeme among those which are valid continuations according to the grammar. Additionally, we impose a maximum syntax tree depth of 5, beyond which a closing bracket is always given priority.

LLM-based Generation: Games are generated as a few-shot task. The model is prompted with a system instruction (Listing 1), the full grammar (Appendix B), and the game implementations from Appendix C as examples. The model is instructed to describe the rules of a new game and produce multiple Ludax implementations of increasing complexity; we evaluate only the final game produced. To encourage diversity, each attempt is seeded with a randomly generated and nonsensical game name such "Outstanding Rainbow Spaniel."

GAVEL-like Evaluation: Inspired by **(author?)** [51], we assess each generated game as follows:

1. A game is *playable* if its description compiles and runs without error.
2. For each *playable* game, we run agent-vs-agent playthroughs using a custom JAX implementation of MCTS with UCB1 [26]
3. We compute the following heuristics from these playthroughs:
    - **Balance:** max winrate gap between players

- **Decisiveness:** fraction of non-draw outcomes
- **Completion:** fraction of games reaching a terminal state
- **Agency:** fraction of turns with > 1 legal move
- **Coverage:** fraction of board sites occupied at least once
- **Strategic Depth:** difference in winrate between a stronger MCTS agent and a weaker one (fewer simulations).

The overall "`GAVEL` score" is the harmonic mean of the individual heuristic scores. Games with a `GAVEL` score > 0.4 are deemed potentially *interesting*. We note that this experiment is preliminary: it omits diversity measures, and the limited search budget for MCTS means they will frequently miss good moves a stronger agent might find. Nevertheless, the fact that an LLM can implement novel games in `Ludax` without finetuning suggests that `Ludax` 's grammar is intuitive and highlights its potential for game generation.

Hyperparameters: For each method, we sample 100 games. For the LLM-based methods, we use a sampling temperature of 0.2. To compute the evaluation score, we run 100 agent-vs-agent simulations for each game. The MCTS agents perform 100 iterations (i.e. traversal, expansion, and random rollout) for each action. For the "strategic depth" evaluation we compare against an MCTS agent that performs 50 iterations per action.

Table 1: GAVEL-based evaluation metrics for 100 generated games, obtained either by uniform random sampling or an LLM. As a baseline, we report results for all default games in Appendix C. *Playable* and *Interesting* denote percentages over all generated games (*Playable* ≥ *Interesting*). *GAVEL score* and *Strategic Depth* report the median and standard deviation, computed only on playable games.

Method	Playable	Interesting	GAVEL Score	Strategic Depth
Default Games	100%	100%	0.69 ± 0.15	0.66 ± 0.15
Random Sampling	4%	0%	0.00 ± 0.00	0.00 ± 0.00
GPT-OSS-120B	95%	83%	0.59 ± 0.22	0.58 ± 0.17
LLaMa-4-17B	82%	42%	0.49 ± 0.21	0.68 ± 0.23