Ludii Game Logic Guide

Ludii Version 1.3.14

Éric Piette, Cameron Browne and Dennis J. N. J. Soemers

Department of Data Science and Knowledge Engineering (DKE)
Maastricht University
Maastricht, the Netherlands

February 20, 2025

Introduction

This technical report outlines the fundamental workings of the game logic behind Ludii – a "general game system" that can be used to play a wide variety of games. Ludii is a program developed for the ERC-funded Digital Ludeme Project, in which mathematical and computational approaches are used to study how games were played, and spread, throughout history. The source code of Ludii is available at https://github.com/Ludeme/Ludii

This report explains how general game states and equipment are represented in Ludii, and how the rule *ludemes*[1] dictating play are implemented "behind the scenes", giving some insight into the core game logic behind the Ludii general game player. This guide is intended to help game designers using the Ludii game description language (Browne, Soemers, Piette, Stephenson, & Crist, 2020) to understand it more completely and make fuller use of its features when describing their games.

Up-to-date news on Ludii and the Digital Ludeme Project can be found on the relevant websites and Twitter pages:

- **Ludii website**: http://ludii.games/
- **Ludii Twitter**: https://twitter.com/ludiigames
- **Digital Ludeme Project website**: http://www.ludeme.eu/
- **Digital Ludeme Project Twitter**: https://twitter.com/archaeoludology/

Please contact the authors with any comments or corrections at: ludii.games@gmail.com

---

[1]Ludemes are "game memes" or units of game-related information (Parlett, 2016).

Contents

5

1
Overview

Ludii (Piette et al., 2020) is a complete general game system implemented in Java. The objective of this report is to bring to light the core elements of Ludii used for game representation, including the code logic of each class in a game description.

Ludii uses a *class grammar* approach (Browne, 2016) to give a direct correspondence between the keywords in its game descriptions and the underlying Java code that implements them, using Java Reflection. The game description language is generated from the constructors in the class hierarchy of the Ludii source code. Game descriptions expressed in the grammar are automatically instantiated back into the corresponding library code for compilation, giving a guaranteed 1:1 mapping between the source code and the grammar.

The implementation details are hidden from the user, who only sees the simplified grammar, which summarises the code to be called. The full grammar–with full documentation–is available in the Ludii Language Reference, which may be downloaded from https://ludii.games/downloads/LudiiLanguageReference.pdf.

2
Game Representation

2.1 A Ludemic Approach

Ludii uses a *ludemic* approach to game design, first outlined in (Browne, 2008). The core of Ludii is a ludeme library, consisting of a number of classes, each implementing a specific ludeme. A *ludeme* is a conceptual unit of game-related information, which we use to define both the *form* of the game (its rules and equipment) and its *function* (legal moves and outcomes for the end state).

The game description for a Ludii implementation of a game is written in a .lud file, which is a tree of ludemes. Each term in the .lud file corresponds to a class in the core of Ludii or to a parameter used in a class such as a string, an integer, an array or an enumeration type. The language used to describe games for Ludii is automatically derived from the ludeme classes available in Ludii's .jar file. **Ludeme 1** below is an example of the complete Ludii game description for Amazons[2] in the EBNF-style grammar used by Ludii.

---

[2] Amazons: Wikipedia
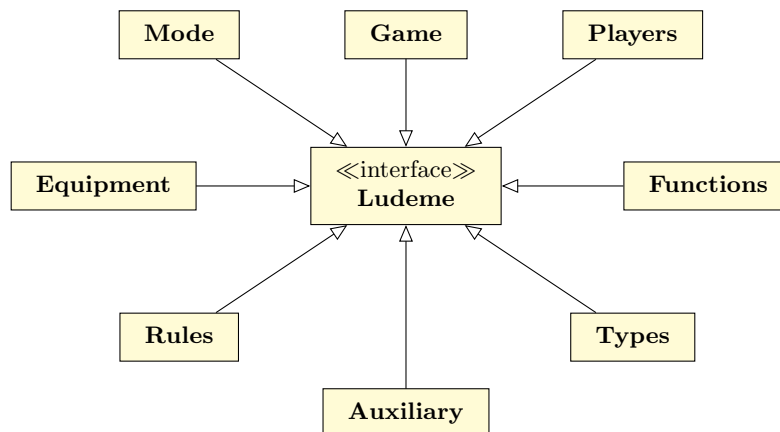
❖ Ludeme 1

```
(game "Amazons"
    (players 2)
    (equipment {
        (board (square 10))
        (piece "Queen" Each (move Slide (then (moveAgain))))
        (piece "Dot" Neutral)
    })
    (rules
        (start {
            (place "Queen1" {"A4" "D1" "G1" "J4"})
            (place "Queen2" {"A7" "D10" "G10" "J7"})
        })

        (play
            (if (is Even (count Moves))
                (forEach Piece)
                (move Shoot (piece "Dot0"))
            )
        )

        (end (if (no Moves Next)  (result Mover Win) ) )
    )
)
```

Each term after an opening parenthesis in this tree of ludemes (for example, players, board and count) is a ludeme implemented in the core of Ludii by a corresponding Java class (for example, Players.java, Board.java, and Count.java, respectively). Terms between quotes (such as "Queen1" or "Queen2") are string parameters, in this case for the ludeme (place ...). The numbers 2 and 10 are integer parameters for ludemes (players ...) and (square ...), respectively. Terms between curly braces '{' and '}' are elements of an array, which can also be a parameter of a ludeme.

Every java class corresponding to a ludeme implements the interface Ludeme.java, which is at the top of the ludeme class hierarchy. They are organised into the following categories:


- **Game**: The root ludeme of any tree of ludemes.

- **Players and Mode**: These two ludemes are direct members of game.
- **Equipment**: Any ludeme that is part of the game, such as the game board, pieces, dice, and cards.
- **Rules**: The ludemes representing starting, playing and ending rules. For example, the Moves ludemes are playing rules.
- **Functions**: The ludemes representing functions used in a game description, such as Boolean functions, Graph functions, Integer functions, Directions functions and Region functions.
- **Types**: This includes ludemes that are enumeration types such as Compass Direction (N, S, W, E ...), Role types (Mover, Next, P1, ...) or Site types (Vertex, Edge, Cell).
- **Auxiliary**: This includes ludemes such as pair used in the Map ludeme to associate two elements or the from/between/to ludeme structures used in many Moves ludemes.

The Types ludemes are described in the Ludii Language Reference (Browne et al., 2020). All the other ludemes are described below in chapters 3, 4 and 5.

2.2 Context

In Ludii, a single object called *Context* is used to store the game $\mathcal{G}$, its game state representation s, and the sequence of moves played so far called the trial τ.
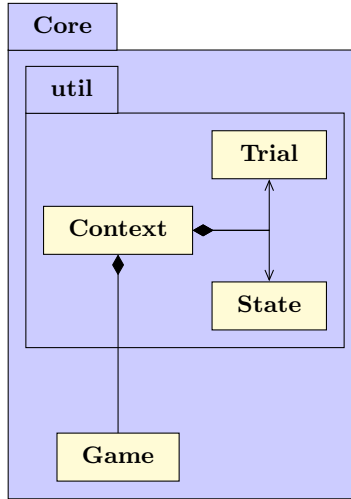
For any operation such as computing the graph of a container, computing the initial state s_0, or computing the legal moves for a state s, Ludii evaluates a tree of ludemes by calling a method *eval(Context context)* to evaluate it according to the current state. The *Context* also contains the random number generator used for any stochastic operations in the corresponding trial and the value of model (Alternating, Simultaneous, or Simulation) used to apply moves or compute the legal moves in a specific game state.

The following subsections describe how a game is represented internally in Ludii.

2.3 Game

In Ludii, a game is defined by a quintuple of ludemes $\mathcal{G} = \langle Name, Players, Mode, Equipment, Rules \rangle$. The Game ludeme (game ...) is the root of a tree of ludemes and corresponds internally in Ludii to the class *Game.java*. The main members of this class correspond to each element of the tuple defining a game:
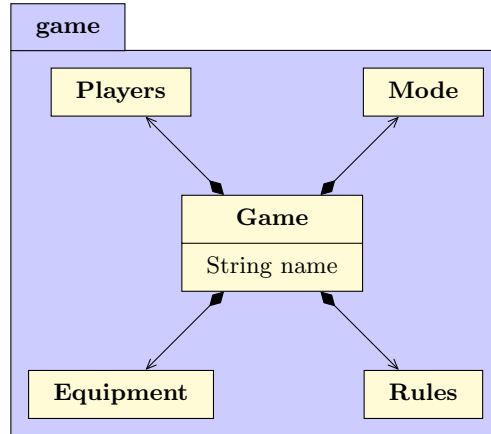
- **Name**: The name of the game.
- **Players**: Defines all information about the players of the game.
- **Mode**: Defines the game control mode.
- **Equipment**: Defines the items for the game.
- **Rules**: Defines the starting, playing and ending rules for the game.

Each Game instance also contains values for the following Boolean variables, which may activate different meta rules:

- **automove**: Activates the meta rule to apply automatically all the legal moves only applicable to a single site (e.g. Trax[3]).
- **gravity**: Activates the meta rule to apply a specific gravity (e.g. Shibumi board[4]).
- **pin**: Activates the meta rule to remove from the legal moves the ones which can remove a piece which is supported by more than one piece (e.g. Shibumi board[5]).
- **repetitionType**: Activates a specific no-repetition rule (e.g. Go[6]).
- **usesSwapRule**: Activates the Swap rule for the game (e.g. Hex[7]).

After compilation of the game, when all the ludemes are instantiated, Ludii computes the game flags by recursively calling the method *computeGameFlags()* on all the ludemes used to describe a game $\mathcal{G}$ and stores them in a long variable called *gameFlags*. These flags are used for a variety of purposes, such as changing how some ludemes are evaluated, calculating the correct state for a game, and optimising compilation time.

---

[3]Trax: Wikipedia
[4]Shibumi games: Shibumi Rule Book
[5]Shibumi games: Shibumi Rule Book
[6]Go: Wikipedia
[7]Hex: Wikipedia

Here is the list of all the game flags:

Game Flag	Description
UsesFromPositions	"On" if this game may generate moves using different 'from' and 'to' positions.
SiteState	"On" if the game involves a state value for a site.
Count	"On" if the game involves a count for a site.
HiddenInfo	"On" if the game has hidden info.
Stacking	"On" if the game is a stacking game.
Boardless	"On" if the game is a boardless game.
Stochastic	"On" if the game involves stochasticity.
DeductionPuzzle	"On" if the game is a deduction puzzle.
Score	"On" if the game involves a score.
Visited	"On" if the game stores all the visited sites in the same turn.
Simultaneous	"On" if the game has simultaneous actions.
ThreeDimensions	"On" if this game is a 3D game.
NotAllPass	"On" if the game does not end if all the players pass.
Card	"On" if the game uses cards.
LargePiece	"On" if the game has one or more large pieces.
SequenceCapture	"On" if the game allows multiple captures in sequence.
Track	"On" if the game uses tracks.
Rotation	"On" if the game uses rotations.
Team	"On" if the game has teams.
Bet	"On" if the game has some betting actions.
HashScores	"On" if the game has some hash scores.
HashAmounts	"On" If the game has some hash amounts.
HashPhases	"On"if the game has some hash phases.
Graph	"On" if the game is a graph game.
Vertex	"On" if the game is played on the vertices.
Cell	"On" if the game is played on the cells.
Edge	"On" if the game can be played on the edges.
Dominoes	"On" if the game has dominoes.
LineOfPlay	"On" if the game has a line of play used.
MoveAgain	"On" if the game has some replay actions.
Value	"On" if the game has some components with a value.
Vote	"On" if the game has some vote actions.
Note	"On" if the game has some Note actions.
Loops	"On" if the game involves a loop.

StepAdjacentDistance	"On" if the adjacent distance between sites is needed.
StepOrthogonalDistan	"On" if the orthogonal distance between sites is needed.
StepDiagonalDistance	"On" if the diagonal distance between sites is needed.
StepOffDistance	"On" if the off diagonal distance between sites is needed.
StepAllDistance	"On" if the distance between all sites is needed.
InternalLoopInTrack	"On" if the tracks on the game have an internal loop.
UsesSwapRule	"On" if the game uses a swap rule.
RepeatInGame	"On" if the game checks the repetition in the game.
RepeatInTurn	"On" if the game checks the repetition in the turn.
PendingValues	"On" if the game uses some pending states/values.
MapValue	"On" if the game uses some maps to values.
RememberingValues	"On" if the game uses some values to remember.

The Game object also contains two variables: maxTurnLimit and maxMovesLimit, used as a limit for the number of turns and a limit for the number of moves of a game, respectively. Currently the limit of moves used in Ludii is 10,000 and the limit of turns is 1,250 multiplied by the number of players. When one of these limits is reached but the game is not over, the game ends in a draw for the remaining players.

2.4 State

A game state s_i corresponds to all the information of the game after i moves played from the initial state s_0. According to the flags of the game $\mathcal{G}$, Ludii automatically selects the appropriate state type to ensure the most suitable representation.

2.4.1 Container State

Each container of a game $\mathcal{G}$ is modelled by a graph defined by a set C of cells, a set V of vertices and a set E of edges. We specify *locations* $loc = \langle c_i, t_i, s_i, l_i \rangle$ by their container $c = \langle C, V, E \rangle$, a type of site $t_i \in \{$Cell, Vertex, Edge$\}$, a site $s_i \geq 0$, and a *level* $l_i \geq 0$. Every location specifies a specific type of site in a specific container at a specific level.

Each container of the game has its own state representation called `ContainerState`. Different representations are implemented in order to minimise the memory footprint and to optimise the time needed to access necessary data for reasoning on any game:

- **ContainerFlatState**: For games played only on the cells.
- **ContainerGraphState**: For games played on the vertices or the edges.
- **ContainerStateStacks**: For stacking games played only on the cells.
- **ContainerGraphStateStacks**: For stacking games played on the vertices or the edges.
- **ContainerStateCards**: For card games.
- **ContainerDeductionPuzzleState**: For deduction puzzles.
- **ContainerDeductionPuzzleStateLarge**: For deduction puzzles with more than 32 possibles values per variable.
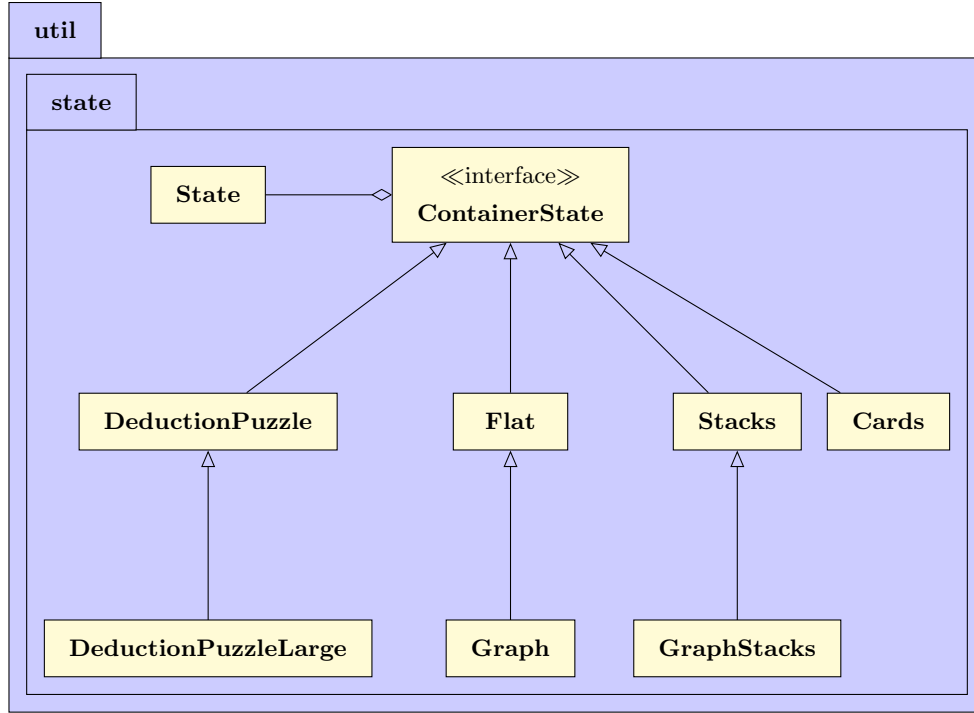
A `ContainerState` *cs* is implemented as a collection of data vectors for each playable site. Each vector is defined using a custom `BitSet` class, called `ChunkSet`, that compresses the required state information into a minimal memory footprint, based on each game's definition. For example, if an N-player game involves no more equipment than a board and uniform pieces in N colours, then the container state is described by a `ChunkSet` subdivided into chunks of B bits per board cell, where B is the lowest power of 2 that provides enough bits to represent every possible state per playable site (including state 0 for the empty sites).[8]

The different possible data vectors are:

- **what**: The index of a component at a specific location, 0 if no component.

- **who**: The index of the owner of a component at a specific location, 0 if no component.

- **count**: The number of the same component at a specific location.

- **state**: The value of the state at a specific location.

- **rotation**: The value for the rotation direction at a specific location, 0 corresponding to the first supported direction by the board in starting with North then incrementing clockwise.

- **value**: The piece value at a specific location.

- **playable**: For boardless games, returning if a location is playable (1) or not (0).

- **hidden[]**: For each player, if the location is invisible.

- **hiddenWhat[]**: For each player, if the piece index on the location is hidden.

- **hiddenWho[]**: For each player, if the piece owner on the location is hidden.

---

[8]Chunk sizes are set to the lowest power of 2 to avoid issues with chunks straddling consecutive `long` values.

- **hiddenCount[]**: For each player, if the number of pieces on the location is hidden.

- **hiddenState[]**: For each player, if state of the location is hidden.

- **hiddenRotation[]**: For each player, if piece rotation is hidden.

- **hiddenValue[]**: For each player, if the piece value is hidden.

These data vectors are instantiated or not according to the game flags. For example the different vectors about hidden information are instantiated only if the game involves hidden information.

On the following picture, the different effects of using the hidden information is shown:

For each line, a king tile owned by the white player is placed on the two cells with the same data. The number in the top left of the tile is the value of the piece. In the second column of each line, a different type of hidden information is shown:

- line 1: All the data are invisible to the white player.

- line 2: The piece index is hidden to the white player.

- line 3: The piece owner is hidden to the white player.

- line 4: The number of pieces is hidden to the white player.

- line 5: The piece value is hidden to the white player.

- line 6: The piece rotation is hidden to the white player.

- line 7: The state of the site is hidden to the white player.

Each of these data has a maximum value. The maximum value of the data vector 'what' corresponds to the number of pieces and the maximum value of the data vertor 'who' corresponds to the number of players + 2 (included a neutral player and a shared player). The maximum value of the data vectors 'count', 'state', 'rotation' and 'value' are computed when the game is computed. The rotation maximum is fixed and corresponds to the maximum number of possible directions of each playable site. The three other data are computed thanks to the initial state but can also be adapted in the piece definition:

- count: The default maximum value is the maximum between the number of playable sites corresponding to the default site type of the board and the sum of all the pieces placed in the initial state. That value can be modified with the (piece ...) ludeme in specifying the parameter "maxCount:".

- state: The default maximum value is the maximum between the number of players, the maximum local state of a piece placed in the initial state, the maximum number of states of a die component and the maximum states of a large piece. That value can be modified with the (piece ...) ludeme in specifying the parameter "maxState:".

- value: The default maximum value is the maximum between the maximum value of a piece placed in the initial state and 128 (corresponding to 2^7). That value can be modified with the (piece ...) ludeme in specifying the parameter "maxValue:".

14

2.4.2 Other data

The information about the player who moved in the previous state ($prev(s)$), the player who has to move in the current state ($mover(s)$) and the player who has to move in the next state ($next(s)$) are stored in 3 variables: prev, mover, next.

Each state s encodes also the following structured data values used to model the game:

- **amount**: Amount, for example of money, belonging to each player.
- **counter**: Variable, incremented by one after each move played.
- **currentDice**: The current dice values of each set of dice.
- **currentPhase**: Current Phase for each player.
- **diceAllEqual**: True if all the values of the dice are equal.
- **isDecided**: The decision taken after voting.
- **notes**: The notes sent to each player during the game.
- **numTurn**: The number of turns played so far.
- **numTurnSamePlayer**: The number of turns played successively by the same player.
- **pendingValues**: Values stored in a set which are in pending between two states.
- **pieceToRemove**: For sequence of capture to remove (e.g. some draughts games[9]).
- **playerOrder**: Gives the current game state player order initially indexed by the original game start player order (which may be different after swaps)..
- **propositions**: The list of propositions done in the previous state.
- **pot**: The value of the pot, for example of money, of the game.
- **remainingDominoes**: All the remaining dominoes.
- **rememberingValues**: All the values stored in the state by previous state in calling the move (remember Value ...).
- **stalemated**: For every player, a bit indicating whether they are stalemated.
- **sumDice**: The sum of all dice.
- **teams**: Team to which each player belongs, if any.
- **triggered**: For every player, a bit indicating whether some conditions are triggered (e.g. checkmate).
- **trumpSuit**: The index of the suit which is the trump in a card game.
- **valueMap**: Values set to custom variables in the description.
- **valuesPlayer**: Values associated with each player.
- **visited**: BitSet used to store all the locations already visited (from and to) by each move done by the player in a sequence of turns played by the same player.
- **votes**: The list of votes taken for the propositions.

---

[9]International Draughts: Wikipedia

2.4.3 Deduction puzzle

In case of a deduction puzzle, the state s of Ludii is modelled as a Constraint Satisfied Problem (CSP) (Piette, Piette, Stephenson, Soemers, & Browne, 2019). A CSP consists of a set of variables – each associated with a domain of possible values – and a set of constraints that link the variables and define allowed combinations of values among them. As an example of a deduction puzzle, we provide the complete description of the 8 Queens Puzzle.[10]

Ludeme 2
```
(game "N Queens"
    (players 1)
    (equipment  {
        (puzzleBoard (square 8) (values Cell (range 0 1)))
        (regions {AllDirections})
    })

    (rules
        (play
            (satisfy {
                (is Count (sites Board) of:1 8)
                (all Different except:0)
            })
        )
        (end (if (is Solved) (result P1 Win)))
    )
)
```

Each site on the board is seen as a variable and the domain of each variable is defined thanks to the ludeme (*values* ...). In the case of the 8 Queens Puzzle, 64 variables represent the cells of the board with each of them a domain of two values {0 1}. The constraints of the puzzle are defined by the ludeme (satisfy ...).

The dedicated container state for deduction puzzles called *ContainerDeductionPuzzleState* models each variable as a `BitSet` where each bit corresponds to one single possible value for this variable. Initially all these bits are set to true to mean they are still possible values for this variable. To remove a value from the set of possible values, the bit must be false. To set a value to a variable, only one single bit has to be true. When all the variables are set to a value, the ludeme (is Solved ...) checks if all the constraints are satisfied. If this is the case the player wins.

The main methods of this container state are:

- **bit(var,value)**: True if the value is a possible value for the variable *var*.

- **set(var,value)**: To keep only the bit corresponding to that value true and set all the others to false for the variable *var*.

- **toggle(var,value)**: To toggle the bit corresponding to that value for the variable *var*.

- **reset(var)**: To reset all the bits of each value of a variable *var* to true.

- **isResolved(var)**: True if a value is set to the variable *var*.

- **what(var)**: The value set to the variable *var*. If the value is not set, it returns 0.

---

[10]8 Queens Puzzle: Wikipedia

2.5 Trials

A Ludii successor function is given by $\mathcal{T} : (S \setminus S_{ter}, \mathcal{A}) \mapsto S$, where S is the set of all the Ludii game states, S_{ter} the set of all the terminal states, and $\mathcal{A}$ the set of all possible lists of actions.

Given a current state $s \in S \setminus S_{ter}$, and a list of actions $A = [a_i] \in \mathcal{A}$, $\mathcal{T}$ computes a successor state $s' \in S$. Intuitively, a complete list of actions A can be understood as a single "move" selected by a player, which may have multiple effects on a game state (each implemented by a different primitive action).

A trial τ is a sequence of states s_i and action list A_i: $\langle s_0, A_1, s_1, \ldots, s_{f-1}, A_f, s_f \rangle$ such that $f \geq 0$, and for all $i \in \{1, \ldots f\}$,

- the played action list A_i is legal for the $mover(s_{i-1})$

- states are updated: $s_i = \mathcal{T}(s_{i-1}, A_i)$

- only s_f may be terminal: $\{s_0, \ldots, s_{f-1}\} \cap S_{ter} = \emptyset$

τ is stored in a `Trial` object, providing a complete record of a game played from start to end, including all the moves made stored in a list. Thanks to that any reasoning on any game can be parallelised using separate trials per thread. All the data members of the `Game` object are constant and can therefore be shared between threads. A thread will be able to use a `Trial` object to compute any playouts from any state.

A trial is over when the variable status is set to the index of the winner of the game (or to 0 in case of a draw) and the outcome of the game corresponding to the ranking of the players is stored in an array of double values called ranking. The trial also stores the list of all the locations of each component placed in the initial state in order to be able to access them with the ludeme (sites Start ...) and a list of long values encoding the previous states thanks to a Zobrist hashing in order to be able to compare efficiently two states in case of meta rules such as no repetition.

When a new state s_{i+1} is reached after applying a move selected from the list of legal moves for a state s_i, Ludii computes the new list of legal moves of s_{i+1} and stores them in the trial in order for any feature to access them quickly without needing to compute them again.

2.5.1 Move and Actions

As described in the previous section, the transition between two successive states s_i and s_{i+1} is possible thanks to a sequence of actions A_i applied on s_i. In Ludii, such a sequence is modelled as a Move object.
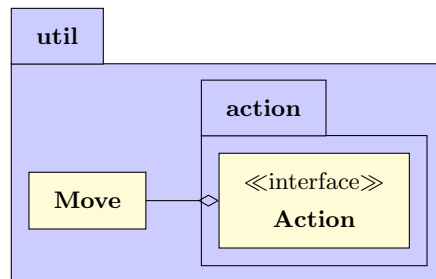
In Ludii, an atomic action is the only way to modify the state after its creation. Consequently, when a player selects their move m in the list of legal moves available in the trial, Ludii applies successively each atomic action in the list of actions composing the move m.

Each action and move are associated with their own data:

- **from**: The 'from' site. If the action is not related to a site, that value is equal to -1.

- **levelFrom**: The level of the 'from' site. If the game is not a stacking game, that value is equal to 0.

- **fromType**: The graph type of the 'from' site (Cell, Vertex or Edge).

- **to**: The 'to' site. If the action is not related to a site, that value is equal to -1. Moreover, if the action is relative to only one site, that value is equal to the 'from' site.

- **levelTo**: The level of the 'to' site. If the game is not a stacking game, the value is equal to 0.

- **toType**: The graph type of the 'to' site (Cell, Vertex or Edge).

- **stacking**: True if the move is related to a stack.

- **oriented**: True if the action is oriented.

- **what**: The index of the component moved.

- **who**: The index of the player making a move.

- **state**: The new state value of the site. If the action is not modifying the state value, the value is equal to -1.

- **rotation**: The new rotation value of the site. If the action is not modifying the rotation value, the value is equal to -1.

- **value**: The new piece value of the site. If the action is not modifying the piece value, the value is equal to -1.

Ludii makes a clear difference between a decision move and an effect move. A decision move is the move decided by the player as opposed to an effect move, which is applied following a decision. In each sequence of actions A composing a move m in the list of legal moves, only one action a_d is a decision: $\exists! \ a_d \in A$. For a specific move m all the data defined previously correspond to the data of a_d.

Here is the list of all the current atomic actions designed in Ludii:

Action	Description
ActionAdd	Adds one or more piece(s) to a location.
ActionAddPlayerToTeam	Adds a player to a team.
ActionBet	Makes a bet for a player.
ActionCopy	Copies a component from a location to another.
ActionForfeit	Forfeits a player.
ActionForgetValue	Forget a value previously remembered.
ActionInsert	Inserts a component inside a stack.
ActionNextInstance	Moves the state on to the next instance in a Match.
ActionNote	Sends a message to a player.
ActionMove	Moves a piece from a location to another (only one piece or one full stack).

ActionMoveN	Moves many pieces from a location to another (but not in a stack).
ActionPass	Pass the turn.
ActionPromote	Promotes a piece to another piece.
ActionPropose	Proposes a subject to vote.
ActionRememberValue	Remember a value.
ActionRemove	Removes one or more component(s) from a location.
ActionReset	Resets all the values of a variable to not set in a deduction puzzle.
ActionSelect	Selects a 'from' and 'to' site.
ActionSet	Sets a value to a variable in a deduction puzzle.
ActionSetAmount	Sets a player's amount.
ActionSetCost	Sets the cost of a graph element.
ActionSetCount	Sets the count of a location.
ActionSetCounter	Sets the counter of the state.
ActionSetDiceAllEqual	Specifies for the state that all dice are equal.
ActionSetHidden	Makes a site invisible or not to a player.
ActionSetHiddenCount	Makes the number of pieces on a site hidden or not to a player.
ActionSetHiddenRotation	Makes the piece rotation on a site hidden or not to a player.
ActionSetHiddenState	Makes the state of the site hidden or not to a player.
ActionSetHiddenValue	Makes the piece value on a site hidden or not to a player.
ActionSetHiddenWhat	Makes the piece index on a site hidden or not to a player.
ActionSetHiddenWho	Makes the piece owner on a site hidden or not to a player.
ActionSetNextPlayer	Sets the next player.
ActionSetPending	Sets a value to pending in the state.
ActionSetPhase	Sets the phase of a graph element.
ActionSetPot	Sets the pot.
ActionSetRotation	Sets the rotation value of a location.
ActionSetScore	Sets the score of a player.
ActionSetState	Sets the state of a location.
ActionSetTrumpSuit	Sets the trump suit for card games.
ActionSetValueOfPlayer	Sets the value of a player in the state.
ActionSetVar	Sets a var in the state.
ActionStackMove	Moves a part of a stack.
ActionStoreStateInContext	Stores the current state of the game.
ActionSwap	Swap two players.

|	
ActionToggle	Excludes a value from the possible values of a variable in a deduction puzzle.
ActionTrigger	Triggers an event for a player.
ActionUpdateDice	Sets the state of the site where the die is and updates the value of the die.
ActionUseDie	Uses a die and removes it from the current dice of the context.
ActionVote	Votes on a proposition done previously.

3
Equipment

All the items used in a game are defined in the equipment ludeme. Here is the description of an example equipment:

> **⌘ Ludeme 3**
>
> ```
> (equipment {
>         (board
>             (square 5)
>             {
>             (track "Track1" "2,E,N,W,S,E1,N3,E2" P1 directed:true)
>             (track "Track2" "22,W,S,E,N,W1,S3,W2" P2 directed:true)
>             }
>         )
>         (piece "Ball" Each)
>         (map "StartingPoint" {(pair 1 2) (pair 2 22)})
>         (dice d:6 num:2)
>         (regions P1 (sites Bottom))
>         (regions P2 (sites Top))
>     })
> ```

The equipment is defined by a list of item objects, which can be of different types:
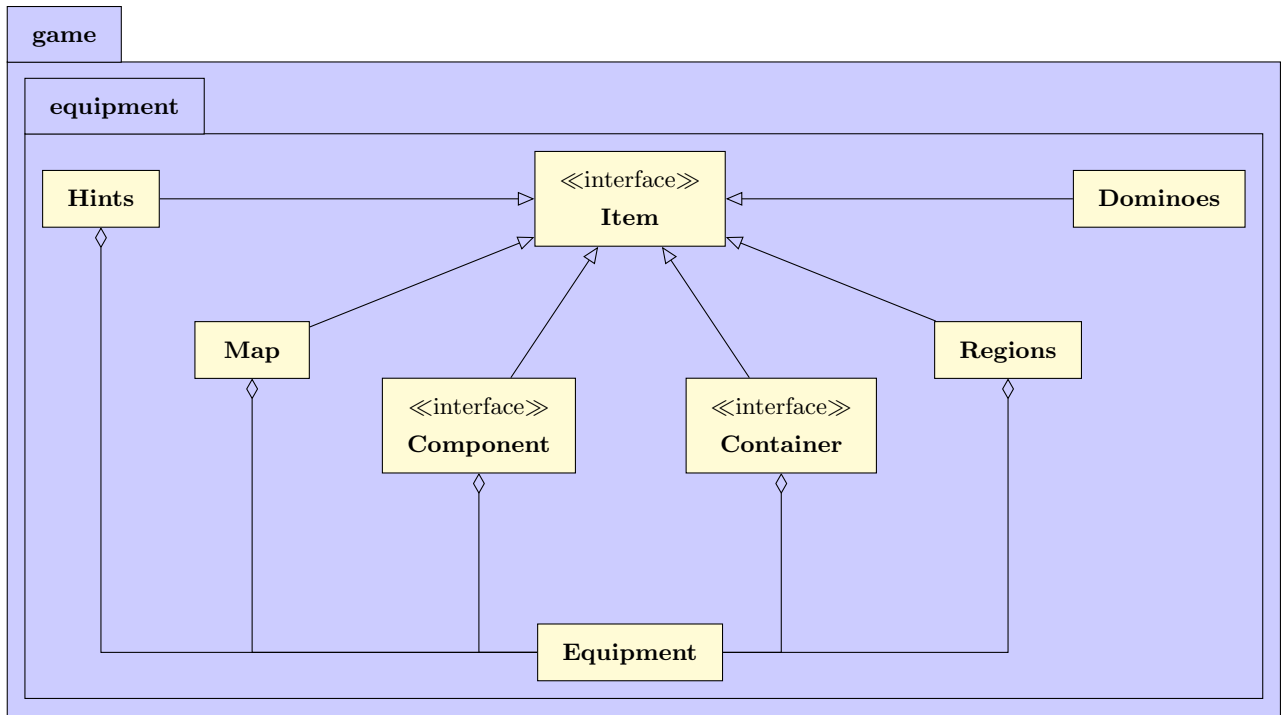
- **Container**: Containers of the game such as the board or the hands of the players.
- **Component**: Components of the game such as the pieces or the tiles.
- **Map**: List of pairs associating two integers.
- **Regions**: Specific regions of the board possibly associated with a name or a player.
- **Dominoes**: Set of dominoes.
- **Hints**: List of hints used for deduction puzzles.

Each type of item is stored in a separate list in the Equipment class in order to access them efficiently. Each item has a name, an owner (equals to -1 if not necessary) and an index used to identify them in the appropriate list in the equipment. In this chapter, we dedicate a section to describe each type of Item object.

3.1 Container

Any game compiled by Ludii has at least a board and potentially other containers used to define the playable sites of the game and the size of each `ChunkSet` used to defined the container state (see Section 2.4.1). Four container types exist:
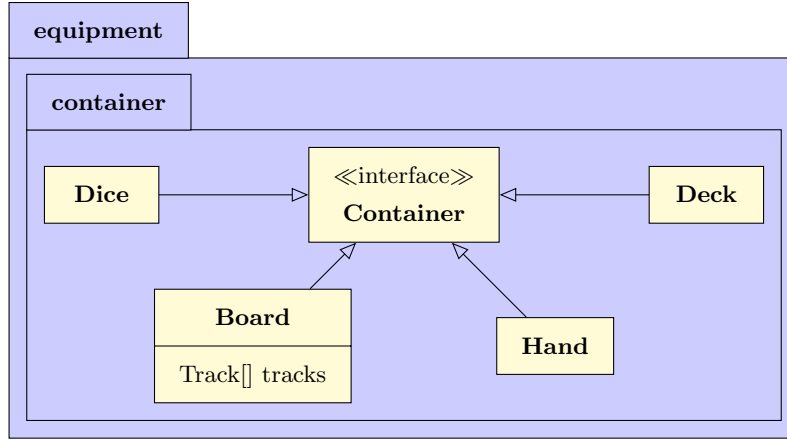
- **Board**: The main board of the game.

- **Hand**: A hand of a player designed as a succession of cells generally used to place the pieces belonging to the player but not currently on the board.

- **Dice**: The set of dice for the game.

- **Deck**: The deck of cards for the game.

Each container of a game $\mathcal{G}$ is modelled by a graph defined by a set C of cells, a set V of vertices and a set E of edges. A Topology object is stored in each Container object and contains a separate list for each graph element type stored as a TopologyElement list (cells, vertices and edges). The topology also contains many pre-generated sets of sites for each graph element type. Here is the list of all the pre-generated sets:

Pre-generated sets	Description
Major	The list of major generator sites.
Minor	The list of minor generator sites.
Corners	The list of corner sites.
Convex Corners	The list of convex corner sites.
Concave Corners	The list of concave corner sites.
Outer	The list of outer sites.
Perimeter	The list of perimeter sites.
Inner	The list of inner sites.
interlayer	The list of interlayer sites.
top	The list of sites in the top of the container.
bottom	The list of sites in the bottom of the container.
left	The list of sites in the left of the container.
right	The list of sites in the right of the container.
centre	The list of sites in the centre of the container.
columns	The list of each column.

rows	The list of each row.
phases	The list of each set of sites in the same phase.
sides	The list of set of sites for each side of the board indexing by direction.
layers	The list of set of sites in the same layer (for 3D games).
axials	The list of axial edges.
horizontal	The list of horizontal edges.
vertical	The list of vertical edges.
angled	The list of angled edges.
slash	The list of angled edges as a slash.
slosh	The list of angled edges as a slosh.

The geometry of each board is described in the next section.

3.1.1 Board Geometry

The board shared by all players is represented internally as a graph, in which the vertices, edges and faces represent the playable *sites* at which components can be placed. The edges represent adjacency between neighbouring vertices and faces. To generate the graph defining the topology of each container, Ludii uses a set of functions called GraphFunctions.

3.1.1.1 GraphFunction

The simplest graph function is the (graph ...) ludeme shown below, which allows the user to specify a Cartesian (x,y) position for each vertex, and then specify the edges of the graph as a list of pairs of adjacent vertex indices. Users can also optionally define cells in this step as a list of lists of vertex indices.

```
✂ Ludeme 4
(board
    (graph
        vertices:{{0 0}{0 1}
            {1 1}{1 0}{0.5 1.5}}
        edges:{{0 1}{1 2}{2 3}
            {3 0}{1 4}{4 2}}
    )
    use:Vertex
)
```
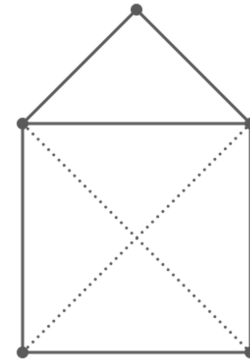
The (board ...) ludeme shown above generates a board on which pieces are played on the vertices, due to the "use:Vertex" parameter. Diagonals within each cell (dotted lines) are automatically generated. If "use:Cell" is specified instead, or no "use:" value is specified, then the game is played on the cells and the following board is generated. Games can also be played on the edges, or any combination of these three graph elements.

> ✂ **Ludeme 5**
> ```
> (board
>     (graph
>         vertices:{{0 0}{0 1}
>             {1 1}{1 0}{0.5 1.5}}
>         edges:{{0 1}{1 2}{2 3}
>             {3 0}{1 4}{4 2}}
>     )
>     use:Cell
> )
> ```
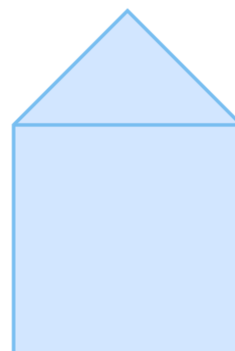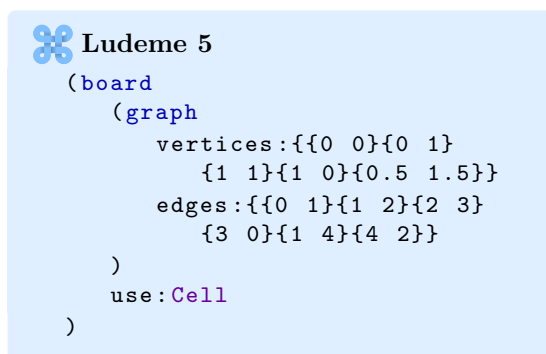
As it can become tedious to specify large numbers of exact vertex positions and edge pairs for more complex boards, a range of graph *generators* and *operators* are provided, as follows.

Graph Generators: These generate a base graph according to a specified basis and/or shape. The following bases are supported.

Graph Bases	Description
(brick ...)	Brickwork tiling.
(celtic ...)	Celtic knotwork tiling.
(hex ...)	Regular hexagonal basis.
(mesh ...)	Arbitrary 2D mesh.
(morris ...)	Morris board designs.
(quadhex ...)	Six-sided quadrilateral mesh, as used for Three-Player Chess.
(square ...)	Regular square tiling.
(tiling ⟨*TilingType*⟩ ...)	Semi-regular tiling.
(tri ...)	Regular triangular tiling.

The following semi-regular tilings are supported.

Semi-Regular Tilings	Description
T488	Semi-regular tiling made up of octagons with squares in the interstitial gaps.
T3464	Rhombitrihexahedral tiling (e.g. Kensington).
T3636	Semi-regular tiling made up of triangles and hexagons.
T4612	Semi-regular tiling made up of squares, hexagons and dodecagons.
T31212	Semi-regular tiling made up of triangles and dodecagons.
T33336	Semi-regular tiling made up of triangles around hexagons.
T33344	Semi-regular tiling made up of alternating rows of squares and triangles.
T33434	Semi-regular tiling made up of squares and pairs of triangles.

T333333_3343	Tiling 3.3.3.3.3.3,3.3.4.3.4.

For example, the following ludeme defines a rhombitrihexahedral (T3464) graph with two major units per side, as used in the Kensington board.

> ⌘ **Ludeme 6**
> `(graph (tiling T3464 2))`
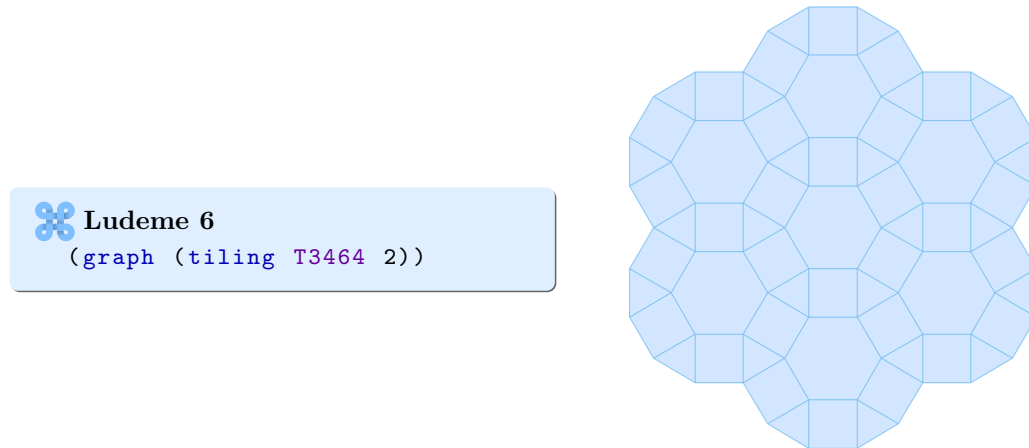
The following board shapes are supported. Note that some shapes are assumed as defaults for certain bases, e.g. the T3434 tiling assumes a hexagonal shape unless otherwise specified, as shown in the above figure.

Graph Shapes	Description
(circle ...)	Circular boards.
(rectangle ...)	Rectangular boards.
(repeat ...)	Boards composed of repeated user-defined shapes.
(shape ...)	Special shapes including Star shaped boards.
(spiral ...)	Spiral boards.
(wedge ...)	Wedge-shapes sub-board, e.g. for adding external "arms" to Alquerque-style boards.

Graph Operators: These modify an existing graph in specific ways. The following operators are provided.

Graph Operators	Description
(add ...)	Add vertices, edges or faces to a graph.
(clip ...)	Clip elements from a graph.
(complete ...)	Join all vertex pairs with an edge.
(dual ...)	Replace the graph with its weak dual.
(hole ...)	Remove internal elements of a graph.
(intersect ...)	Find the intersection of two graphs.
(keep ...)	Keep some elements from a graph and discard the rest.

(layers ...)	Create a 3D graph with layers in the z direction.
(makeFaces ...)	Generate all non-overlapping faces.
(merge ...)	Merge two graphs, keeping all elements of both except where coincident elements are merged.
(recoordinate ...)	Regenerate the coordinate labels for the graph elements.
(remove ...)	Remove specified elements from a graph.
(renumber ...)	Renumber the graph elements (from bottom left rightwards and upwards).
(rotate ...)	Rotate the vertices of a graph a specified number of degrees.
(scale ...)	Scale the vertex positions of a graph in x, y and/or z directions.
(shift ...)	Translate the vertices of a graph in x, y and/or z directions.
(skew ...)	Skew a graph by a specified amount.
(splitCrossings ...)	Split edge crossings into multiple edges that share a vertex at the crossing point.
(subdivide ...)	Subdivide certain faces within a graph.
(trim ...)	Trim orphan elements from the graph exterior.
(union ...)	Union of two graphs, keeping all elements of both.

Graph operators allow complex board topologies to be easily constructed from simple descriptions. For example, the rhombitrihexahedral board shown above can have cells with six or more sides subdivided...

⌘ Ludeme 7
```
(graph
  (subdivide
    (tiling T3464 2)
    min:6
  )
)
```
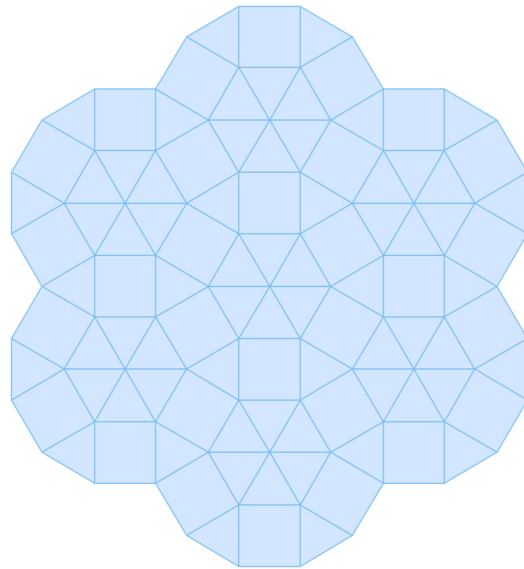
...then be replaced by the weak dual[11] of the resulting graph...

---

[11]The *weak dual* of a graph is the graph induced by creating a vertex at the centroid of each face and connecting each pair of vertices generated by adjacent faces with an edge.

⌘ Ludeme 8

```
(graph
  (dual
    (subdivide
      (tiling T3464 2)
      min:6
    )
  )
)
```
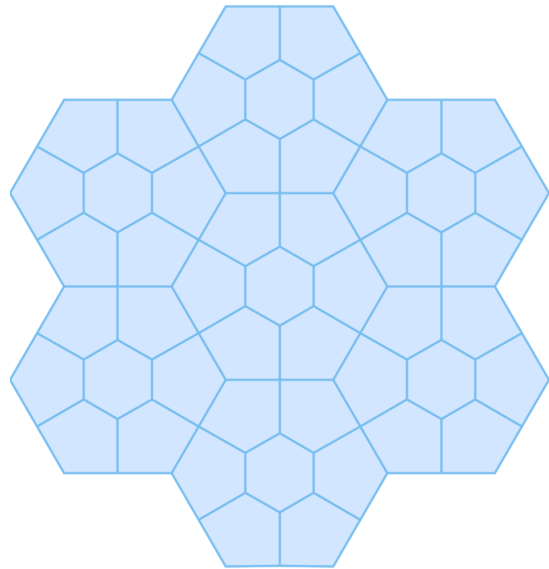

...then have all cells subdivided...

⌘ Ludeme 9

```
(graph
  (subdivide
    (dual
      (subdivide
        (tiling T3464 2)
        min:6
      )
    )
  )
)
```
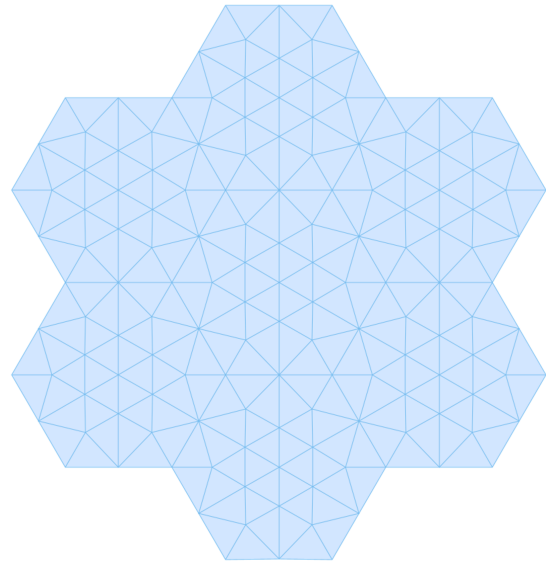

...and again be replaced by the weak dual of the resulting graph. The resulting board is distinct and visually interesting, and not previously used for any game that we are aware of.

Ludeme 10
```
(graph
  (dual
    (subdivide
      (dual
        (subdivide
          (tiling T3464 2)
          min:6
        )
      )
    )
  )
)
```
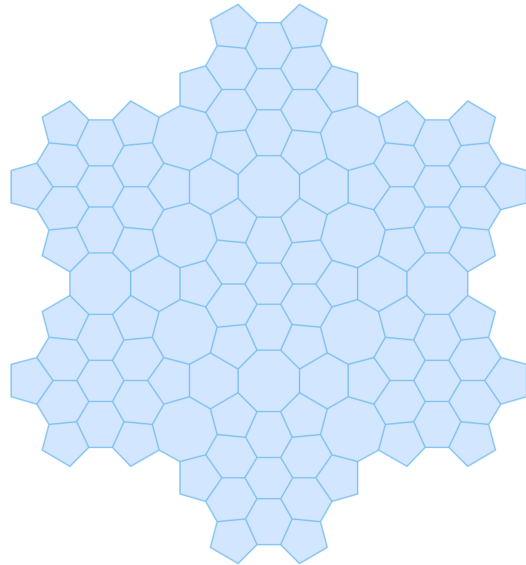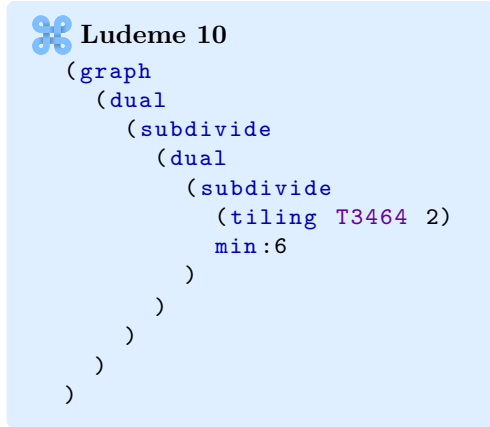

When the game board is generated from a graph produced by a GraphFunction, the following relations are automatically calculated for each site:

- *Steps*: Step to an adjacent element in each direction (if any).

- *Radials*: Contiguous steps to adjacent elements in each direction that minimise change in direction with each step (if any). Radials are used for slide moves, line detection, line-of-sight tests, etc.

More details on using GraphFunctions will be provided in the upcoming paper "General Board Geometry" (in preparation). For the exact syntax of particular GraphFunctions, please see the Ludii Language Reference (Browne et al., 2020).

3.1.1.2 DimFunction And FloatFunction

Any Dimension (e.g. the size of a square) used in entry of all the classes defining the graph functions are DimFunctions classes. Basically, they are used to allow any common mathematical operators between integers such as: Addition, Subtraction, Division, Product, Absolute value, Power, Maximum and Minimum.

Any Float in entry of all the classes defining the graph functions are FloatFunctions. They are also the same operators of the DimFunction but also other such as: Cosine, Sine, Tangent, Exponential, Square Root, Logarithm, Logarithm 10.

3.1.1.3 Direction System

Ludii supports until 16 compass directions following the intercardinal directions[12]:

---

[12]Intercardinal directions: Wikipedia
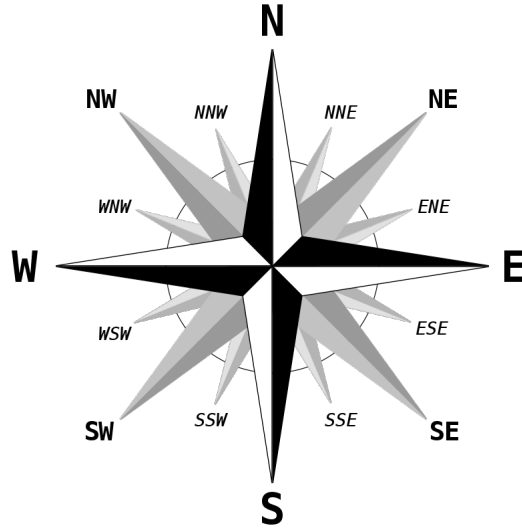
For the containers involving rotational directions defined with a circular ludeme (circle) or a spiral ludeme (spiral), Ludii adds to the compass directions, the four rotational ones: Outer (Out), Inner (In), Clockwise (CW) and Counter Clockwise (CCW). Finally for the 3D games, Ludii handles all the compass directions for the same layer and the 8 most important compass directions for the downwards and upwards directions: D, DN, DNE, DE, DSE, DS, DSW, DW, DNW and U, UN, UNE, UE, USE, US, USW, UW, UNW.

All these directions are defined in different enumeration types called respectively CompassDirection, RotationalDirection and SpatialDirection, and they are all a DirectionFacing object, used to define the direction faced by a piece if necessary. All these facing directions have methods to define the opposite, left and right direction of each of them in order to handle any potential rotation of a piece during the game.
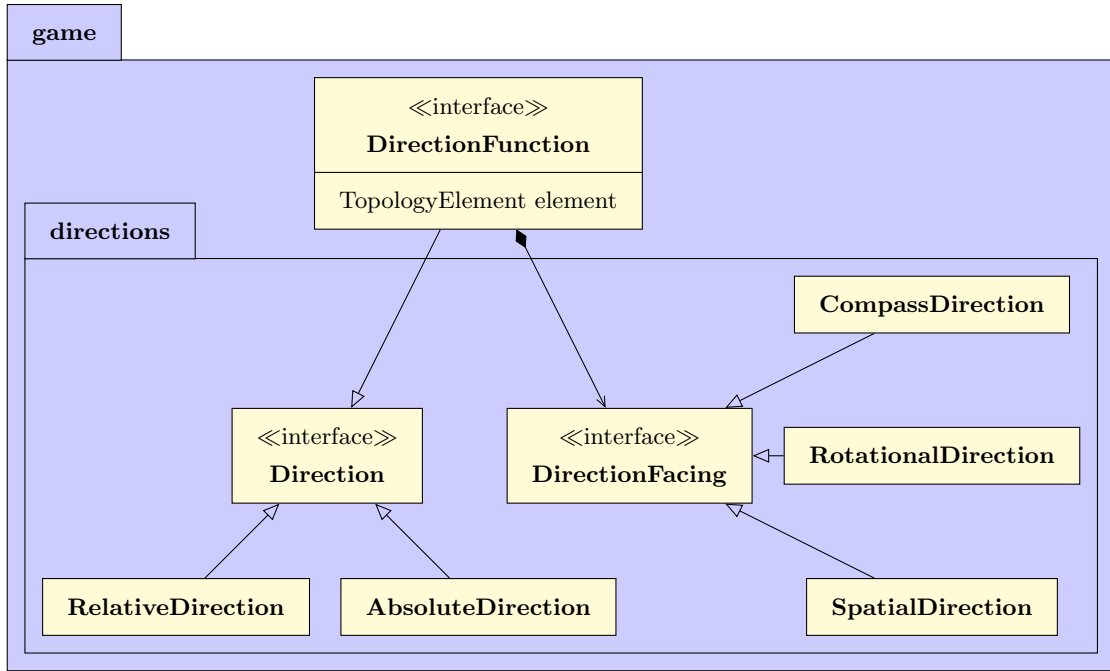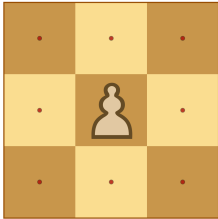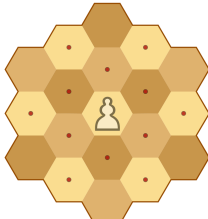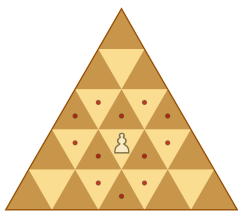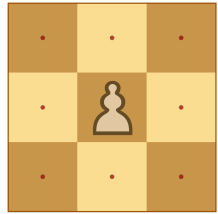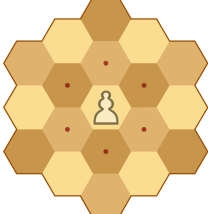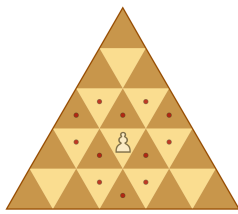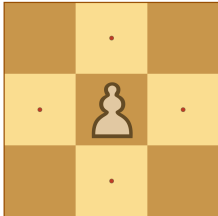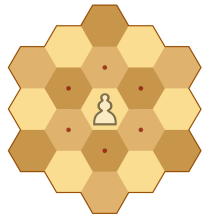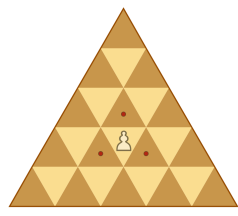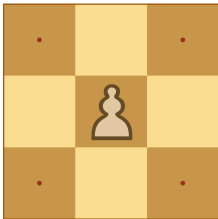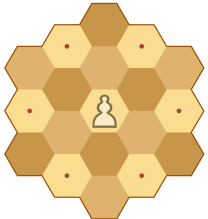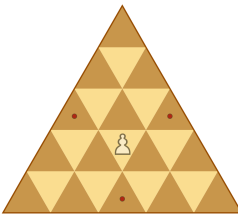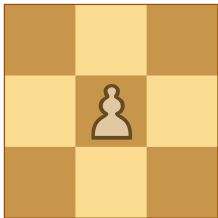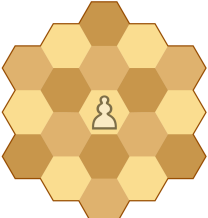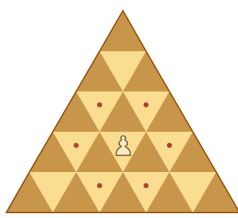
In order to handle any possible direction in a board game, Ludii provides a set of direction functions called DirectionFunction able to return the expecting set of directions according to the context. A direction function can be Absolute in returning constantly the same set of directions of a topology element (playable site) or can be relative and possibly depends on the facing direction of the component placed on this site.

The AbsoluteDirection enumeration type contains equivalents for each possible facing direction and the following ones:

Absolute direction	Description
All	The set of all supported directions of the site.
Angled	All angled directions (NW, NNW, WNW, WSW, SE, SW, SSE, SSW, NNE, ESE, ENE, NE).
Adjacent	All directions between two sites sharing at least one vertex.
Axial	All axial directions (N, S, E, W).
Orthogonal	All directions between two sites sharing at least one edge.
Diagonal	All directions between two sites in line through a vertex.
OffDiagonal	All directions between two cells sharing a vertex but not an edge which are not diagonal.
SameLayer	All directions between two sites in the same layer.
Upward	All directions upward to a site (U, UN, UE, US, UW, UNW, UNE, USW, USE).
Downard	All directions downard to a site (D, DN, DE, DS, DW, DNW, DNE, DSW, DSE).

Rotational	All rotational directions (CW, CCW, IN, OUT).

Here are the different step moves of a piece for the 5 common absolute directions for the 3 common tilings: Square, Hexagonal and Triangle:

Absolute Direction	Square	Hexagonal	Triangle
All			
Adjacent			
Orthogonal			
Diagonal			
OffDiagonal			

The RelativeDirection enumeration type are the following ones:

| Relative direction	Description
Forward	The facing direction.
Backward	The opposite of the facing direction.
Rightward	The most rightward direction between the facing direction and its opposite.
Leftward	The most leftward direction between the facing direction and its opposite.
Forwards	All forward directions between the leftward and the rightward direction.
Backwards	All backward directions between the rightward and the leftward direction.
Rightwards	All rightward directions between the forward and the backward direction.
Leftwards	All leftward directions between the backward and the forward direction.
FL	The direction at the left of the forward direction.
FLL	The second direction at the left of the forward direction.
FLLL	The third direction at the left of the forward direction.
BL	The direction at the left of the backward direction.
BLL	The second direction at the left of the backward direction.
BLLL	The third direction at the left of the backward direction.
FR	The direction at the right of the forward direction.
FRR	The second direction at the right of the forward direction.
FRRR	The third direction at the right of the forward direction.
BR	The direction at the right of the backward direction.
BRR	The second direction at the right of the backward direction.
BRRR	The third direction at the right of the backward direction.
SameDirection	The same direction of the previous move.
OppositeDirection	The opposite direction of the previous move.

Here are the different step moves of a piece facing to the north for the common relative directions for the 3 common tilings: Square, Hexagonal and Triangle:

Relative Direction	Square	Hexagonal	Triangle

Forward
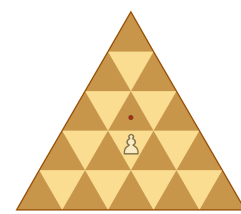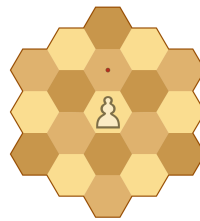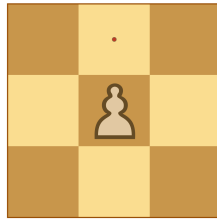
Backward
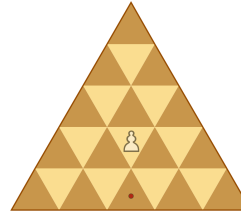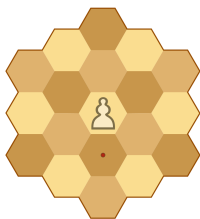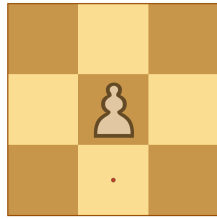
Leftward
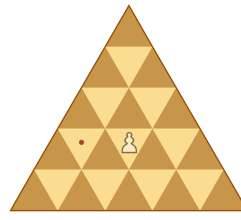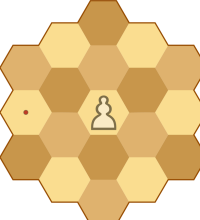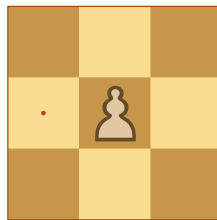
Rightward
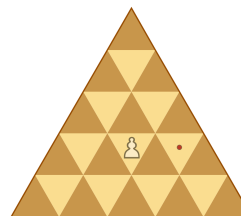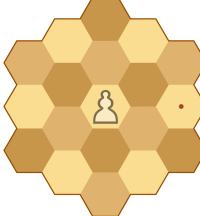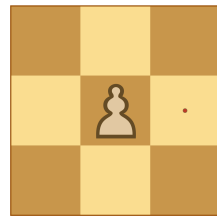
Forwards
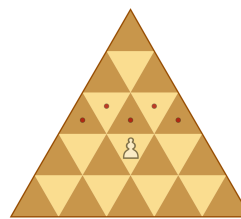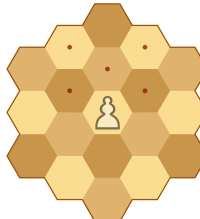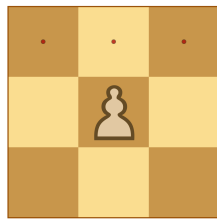
Backwards
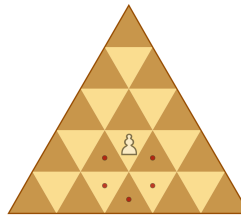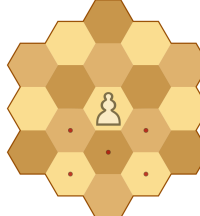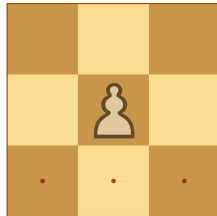
Leftwards
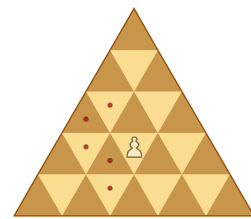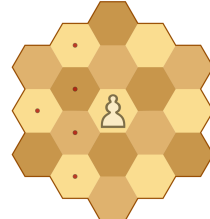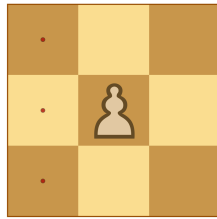
Rightwards
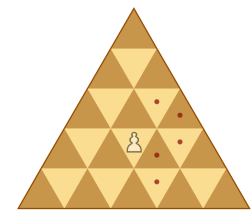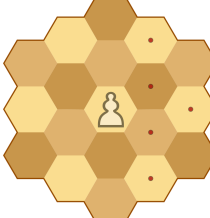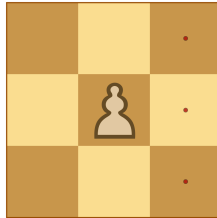
FR
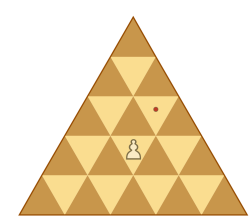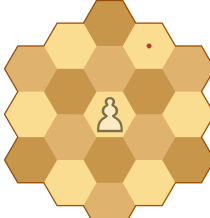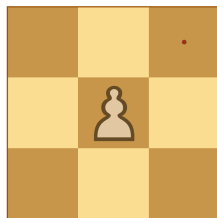
FL
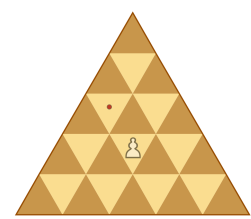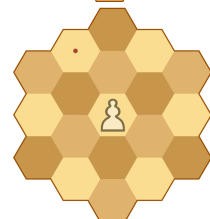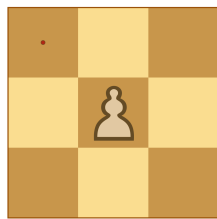
BR
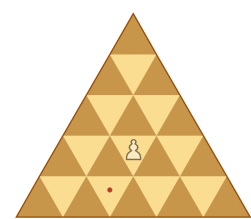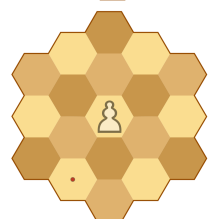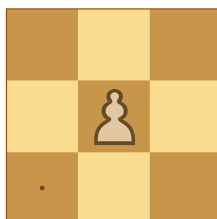
BL
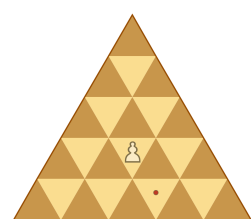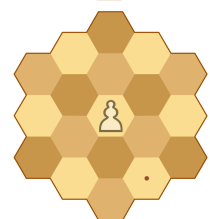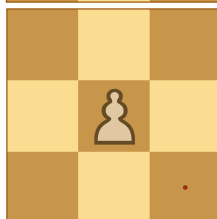
3.1.2 Board

The main board of the game is defined by the ludeme (board ...). Firstly, the board evaluates the GraphFunction defining the graph of the board and converts it into the Topology object by associating each pre-generation related to each graph element. Then, according to the flags of the game, more pre-generations are realised, such as the computation of the minimal distance between two graph elements of the same type.

The parameter "use" of the board ludeme is used to define the default graph element used in all the ludemes involved in the game if not specified. If that parameter is not defined, the default site type is cells. Moreover, the dimensions of a board also depend on that parameter. For example, a Chess board has a dimension of 8 because it is played on the cells but a Go board has a dimension on 19 because it is played on the vertices (even if it has 18 cells by row and column). The Board class is also the only link between the logic part of Ludii and the Graphical User Interface in setting the default design of the board according to the default site if not modified by the metadata of the game.

3.1.2.1 Track

For games involving tracks such as race games, the board builds them. Two ways exist to define a track. One consists of simply listing all the sites from the start to the end. The other one consists of given the starting site of the track then describing in a string the direction and distance of each step of the track. Here is an example of the description of the board of the game "20 Squares"[13] with tracks defined using the second method:

� Ludeme 11

```
(board
    (merge
        (rectangle 3 4)
        (shift 0 1 (rectangle 1 12))
    )
    {
    (track "Track1" "3,W,N1,E,End" P1 directed:true)
    (track "Track2" "12,W,S1,E,End" P2 directed:true)
    }
)
```
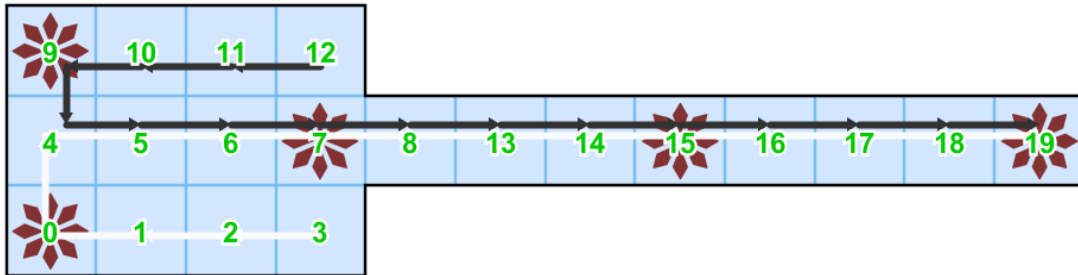

"Track"1 is defined by "3,W,N1,E,End". The board builds that track by beginning it with Cell 3, then, because of the next step "W", all the sites following the radial starting from Cell 3

---

[13]20 Squares: BGG

to the West (in this order: Cells 2, 1 and 0). Then from the last site reached, because of "N1" the next cell is the cell(s) in the radial starting from Cell 0 to the north at distance 1 max (Cell 4). The Same idea applies for the next step describing by "E" in following the radial starting from Cell 4 to the East (in this order: Cells 5, 6, 7, 8, 13, 14, 15, 16, 17, 18, 19). To finish, the last step is "End" meaning that the track ends at an imaginary site after the last site of the track. This "End" feature is useful for escape race games in which a piece can escape the board by moving one more site after the end of the track. Following that method, the track ordering by the cells, is $\{3\ 2\ 1\ 0\ 4\ 5\ 6\ 7\ 8\ 13\ 14\ 15\ 16\ 17\ 18\ 19\ End\}$.

A track can be owned by a player in order to have access to all the tracks used only by a specific player (e.g. "Track1" is owned by P1 (white) and "Track2" is owned by P2 (black) in 20 Squares). Two other properties can be defined for a track:

- **directed**: The track is ordered and it is not possible to move backward in it.

- **loop**: The last site of the track is following by the starting site.

The compiler takes into account two special keywords to make the description more concise, which are:

- End: An imaginary site one step after the end of the track (its hidden value is -2).

- OFF: A value returned by any IntFunction an expected site does not exist (its hidden value is -1). The value UNDEFINED can also be used for that.

Moreover, it is important to note that it is also possible to describe a track beginning in a container other than the board, such as the hand of a player, in listing the expecting sites.

3.1.2.2 Special cases

It is possible to define the board ludeme with ranges of values for each graph element. These data are dedicated to deduction puzzles for defining the range of possible values for each variable and will adapt the state of the game as explained in Section 2.4.3.

The ludeme Boardless is a special type of "board" extending the Board class and defined only by some regular tiling: Square, Hexagonal or Triangle. In its current implementation a fake board of size 41 following the specified tiling will be generated. However, we plan to improve this in the future by building the board only when the sites are necessary in the following states of the game.

In order to improve the description of specific boards, Ludii proposes two specific ludemes:

- **MancalaBoard**: To define any mancala board by giving only the number of rows and columns and the type of stores used: None, Outer, Inner.

- **SurakartaBoard**: To define a board similar to the board of Surakarta for any Graph-Function in specifing the number of loops and the starting row to start the loops from.
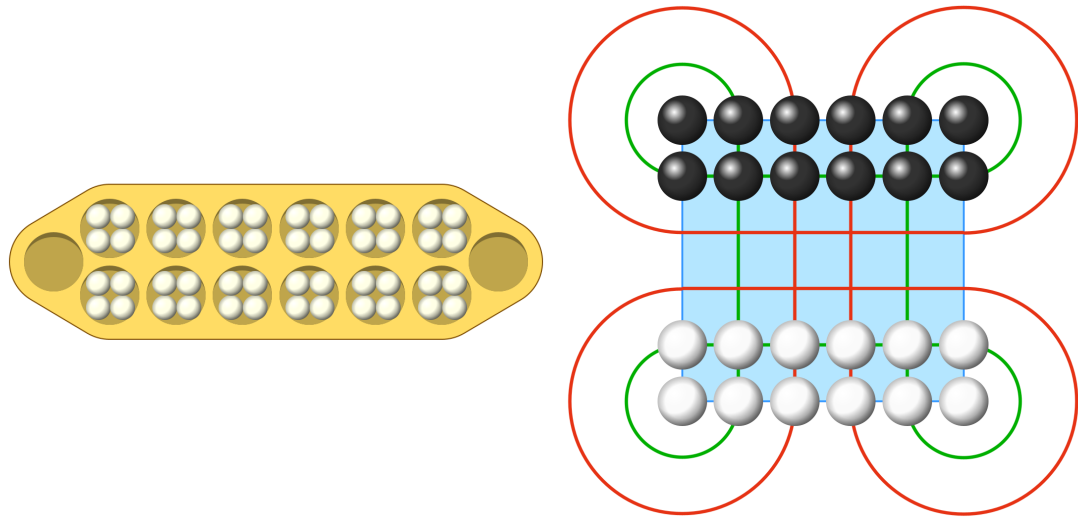
3.1.3   Hand, Dice and Deck

Ludii uses 3 other kinds of container ludemes.

The hand of a player defined by the (hand ...) ludeme corresponds simply to a row of square cells defined by the size of each hand. Each hand is owned by a specific player or it is shared between all the players. That's possible to define a hand of the same size for each player in using the ludeme (hand Each size:3), for example.

The dice ludeme is defined with the number of dice, by the number of faces of each die (the default is 6), and the value of each face of each die (if not defined, from 1 to the number of faces). The dice container is a hand including a site for each die. If not owned, the hand is shared among all players. However, when this ludeme is used, a different component called "DieX" (with X the number of dice created so far) is created for each die.

The Deck ludeme is a hand of a single site with a stack of cards modelling a deck of cards. If not owned, that hand is shared between each player. The deck can be customised by just defining the number of cards per suit (default 13) and the number of suits in the deck (default 4) or in defining the deck card by card thanks to the ludeme Card. The ludeme Card defined a card by its rank and its value and potentially its trump rank and trump value. However the cards are still in implementation and that container is susceptible to be modified.

3.2   Components

The components model the different pieces, tiles, dice, dominoes or cards of a game. Each of them are described by different ludemes.
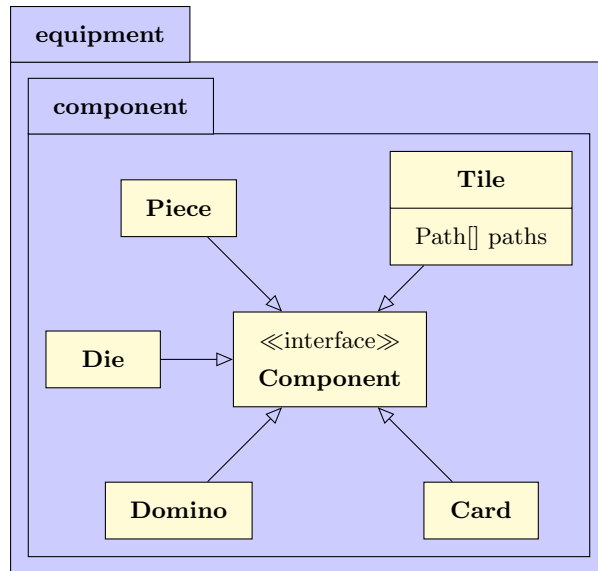
The most used component is described by the Piece class.

🔧 **Ludeme 12**

```
(piece "Ball" Each)
```
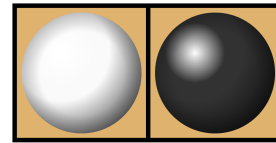
This ludeme generates a component for each player called "Ball". However the name of the component is always its name and the index of the player. For example, for a 2-player game, this ludeme generates a component named "Ball1" for player 1 and a component named "Ball2" for player 2. The parameter "Each" models the owner of the components, if this parameter would be "Neutral" meaning that component is owned by no player, the name of the component would be "Ball0" and if this parameter would be "Shared", meaning this component is owned by all the players, its name would be "Ball" (with no number).

A piece component has many other parameters to specify its properties. As described in Section 3.1.1.3, a facing direction can be set for the piece in order to use the relative directions for the movement of the pieces. Each piece can also be associated with a Moves ludeme called "generator", which will be used to generate some legal moves starting from each site where such a piece is currently placed. More information about how to model moves will be given in Section 4.3.

If a piece can be flipped, the flip values of each face of the piece have to be given (e.g. for Reversi[14]). Each flip value corresponds to the required state value of the site where is placed this piece to be flipped to the other.

---

[14]Reversi: Wikipedia
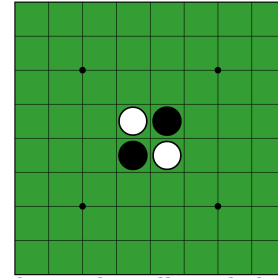
The Tile ludeme is used to specify a component placed only on the cells and shaped like the cells of the board. Some extra parameters can be specified on such components. Firstly, the number of sides of the tile, if not specified, the number of sides is the number of edges of the Cell 0 of the board. Some tiles can have an internal path inside the tile to link some sides to others. Here is an example of the tiles of the game Trax[15]:

⌘ **Ludeme 14**
```
(tile "TileX" numSides:4
    {
    (path from:0 to:2 colour:1)
    (path from:1 to:3 colour:2)
    }
)
(tile "TileCurved" numSides:4
    {
    (path from:0 to:1 colour:1)
    (path from:2 to:3 colour:2)
    }
)
```


The left piece has a path of a colour 1 between the top and the bottom side and a path of colour 2 between the left side and the right side. The value used in the parameter from: and to: are the number of right orthogonal rotations from the top side to get the expected side. For such a square tile, 0 means the top side, 1 the right side, 2 the bottom side and 3 the left side.

Some tiles can be large enough to be placed on many sites simultaneously. Here is an example of L Game[16], a game with large tiles shaped like the letter L.

⌘ **Ludeme 15**
```
(tile "L" Each
    {
    {L F R F F}
    {R F L F F}
    }
)
```
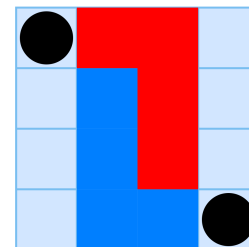

A large tile is defined according to a walk of orthogonal steps. That mechanism is based on Turtle graphics (Wikipedia). In Ludii, a walk starts facing the first orthogonal direction of the board (for a square board, the north), and only 3 parameters describe the walk:

---

[15]Trax: Board Game Geek

[16]L Game: Wikipedia

40

- **F**: Make a forward step in the current facing direction.

- **L**: Modify the facing direction to the left orthogonal direction of the current facing direction.

- **R**: Modify the facing direction to the right orthogonal direction of the current facing direction.

Please note that a walk cannot leave the playing area and return.

For the case of a large tile, Ludii generates all possible sets of sites from a root site for each orthogonal direction and for each list of steps defined in the large tile. For the case of this large tile shaped like the letter L used in a square board, for each site, 8 sets of sites are generated (2 sets of steps and 4 possible orthogonal directions from each site). The state value of the root site is used to select the current set of sites used by the large tile, in the case of the L-shaped large tile, the state value is between 0 and 7. This value corresponds to the number of right rotations from the first orthogonal direction of the cells modulo the number of sets of steps used to define the shape of the large tile. For example, in the picture, the blue large tile rooted from the bottom right site has a state value of 0 and the large tile rooted from the top left site has a state value of 2.

A domino piece is also defined as a large rectangular piece. However, in Ludii a domino uses 8 square sites to be placed (4 sites per side). The reason is because in many dominoes games, it is possible to place a domino in the middle of the long side such as in the next picture. The value of each side of the domino is defined in the Domino ludeme, the first value is allocated to the first 4 sites used by the domino and the second value to the 4 last sites.

> ⌘ **Ludeme 16**
>
> ```
> (domino "Domino45" Shared
>     value:4 value2:5
> )
> ```
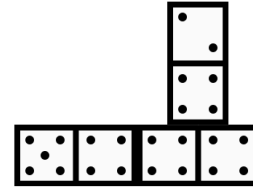
As described in Section 3.1.3, Ludii can generate a set of dice. Each of the dice is a Die component. A die ludeme has a name, an owner and possibily a facing direction and a value such as the piece and tile ludemes. However this component has also a number of faces and different values associated to each face. These data are obtained by using the container Dice as described in the previous section. During the game, at a specific state, when Ludii needs to obtain the value of a die, it uses the state value of the site where the die is placed. For example, a common die with 6 faces starting from the value 1 to the value 6 has: the state 0 corresponding to the value 1, the state 1 to the value 2, ..., the state 5 to the value 6.

> ⌘ **Ludeme 17**
>
> ```
> (dice num:2)
> (dice d:2 from:0 num:4)
> ```
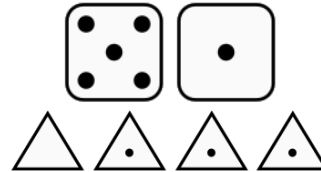
However, to avoid looking at the state value of the sites occupied by the dice at each turn to obtain their values, many shortcuts are implemented in the State class of Ludii (see Section 2.4.2). They provide an efficient way to obtain this information when necessary by using different functions ludemes, which will be described in Chapter 5.

Finally, the last type of component is the Card component. As described in the Section 3.1.3, Ludii generates a set of cards using to the ludeme Deck. Each card is defined by its rank

and its value and potentially its trump rank and trump value. However, the cards are still in implementation and this component is susceptible to be modified.
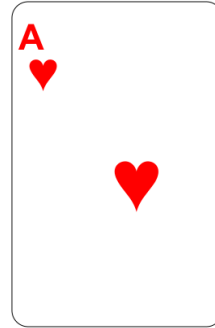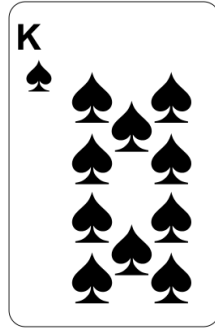
3.3 Other Items

As described in the beginning of this chapter, Ludii contains different other equipment ludemes. This short section describes each of them briefly.

The Dominoes ludeme defines a set of dominoes starting from the Domino with the value 0 for each side to the domino with a specific value for each side. The default maximum value is 6 and, for computational reasons, Ludii limits the value of a side of a domino to 16.

The Hints ludeme defines a set of hints used to solve deduction puzzles. This ludeme is described by an array of Hint ludemes, each of which corresponds to a set of sites and a hint value. These data can be used differently according to the constraints defined in the deduction puzzle. As an example, the puzzle Kakuro[17] at its initial state where only the hints are visible, the corresponding expected sum of the values appear in the left or the bottom of the site where they are placed. Here is the lud description of the rules of the game:


```
Ludeme 18
(rules
    (play
        (satisfy {
            (is Sum (hint))
            (all Different)
        })
    )
    (end
        (if (is Solved)
            (result P1 Win)
        )
    )
)
```

The map equipment is used to create a correspondence between a key and a value. A map is defined by a name and a list of Pair ludemes. The name is optional, if not described, the

---

[17]Kakuro: Wikipedia

default name is "MapX" with X the index of the map in the equipment. A pair ludeme is defined by a key and a value. Whatever the entries, the map ludeme converts them to integers. The entries can be coordinates expressed by a string, RoleType of players or directly using IntFunctions (see Section 5.2) or integers. Here is an example of a map used frequently in race games to obtain the site where the pieces can enter into the board for each player.

> ⌘ **Ludeme 19**
> ```
> (map "EntrySite"
>     {
>     (pair P1 "A3")
>     (pair P2 "A1")
>     }
> )
> ```

The last type of equipment is the Regions ludeme. This ludeme defines specific sets of sites used by the game. A region can be defined by a name and owned by a player. The default name is "RegionX" with X being the index of the owner of the region, and if no owner exists, the first available index. The set of sites belonging to the regions can be defined by a list of RegionFunctions (see Section 5.4) or directly by lists of integers. Here is the description of the home region of each player in English Draughts[18].

> ⌘ **Ludeme 20**
> ```
> (regions "Home" P1 (sites Bottom))
> (regions "Home" P2 (sites Top))
> ```

---

[18]English Draughts: Wikipedia

4
Rules

In Ludii, the rules of a game are divided in 4 types:

- **Meta**: The meta rules of the game applied at each turn whatever the state.

- **Start**: The starting rules defining the initial state.

- **Play**: The playing rules defining the legal moves of the mover in the current state.

- **End**: The ending rules defining the conditions to satisfy to end the game and the consequent outcomes.
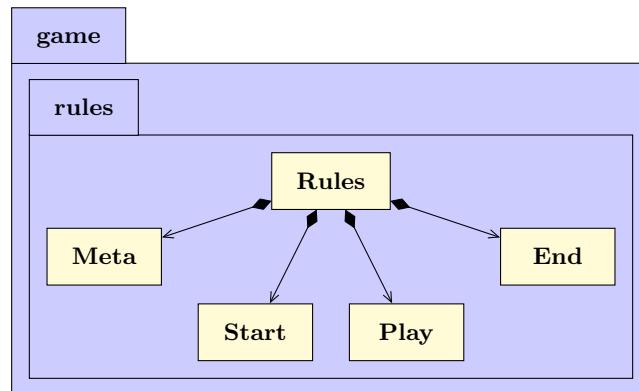
The following sections detail each type of move.

4.1   Meta Rules

A meta rule is a rule applied at each turn whatever the conditions. These rules are optional. Ludii has some meta rules in its current version:

Meta Rule	Description
(automove)	Apply automatically all the legal moves only applicable to a single site.
(gravity ...)	Activates the meta rule to apply a specific gravity.
(passEnd ...)	To modify the end of the game in case of all the players pass their turns.
(pin ...)	Activates the meta rule to remove from the legal moves the ones which can removea piece which is supported by more than one piece.
(swap)	Add the swap move to the legal moves for the first turn of the players different than the first one.
(no Repeat ...)	Filter the legal moves reaching a state already reached before.
(no Suicide ...)	Filter the legal moves reaching a state leading to a direct loss for the mover.

4.2 Start Rules

The starting rules define a list of movements (list of actions) A_0 applied to a state s_{-1} with no piece placed (all the sites are empty in all the containers) and with all the variables set at their default value. After applying A_0 to s_{-1}, the state created is s_0, the initial state of the game. The starting rules are optional, if no starting rules defined, s_0 corresponds to s_{-1}.

Ludii has different starting rules ludemes. The most used is the Place ludeme. This ludeme places piece(s) (or stack of pieces) on a site (or a set of sites). Simultaneously, that's possible to set the state, the rotation value (based on the facing direction of the piece placed on that site) or the value of the pieces placed. Here are some examples of how it's used:

> ❖ **Ludeme 21**
> ```
> (place "Commander1" coord:"A1" rotation:1)
> (place "Marshal1" 84 value:5)
> (place "Pawn1" (sites Row 1))
> (place "Rook2" {"A8" "H8"} state:1)
> (place Stack "Goat0" (sites {"B2" "D2" "B4"}) counts:{5 5 5})
> (place Random {"Ball"} count:29)
> ```

The first line places a piece called "Commander" owned by the player 1 on the site corresponding to the coordinate A1 on the board with one rotation to the right according to its facing direction. The second line places a piece called "Marshall" owned by the player 1 on the site 84 and that site will be invisible to the player 2. The third line places a piece called "Pawn" owned by the player 1 on all the sites of the row of index 1. The fourth line places a piece called "Rook" owned by the player 2 on the sites corresponding to the coordinates A8 and H8 with a state site of 1. The fifth line places a stack of 5 pieces called "Goat" owned by no player (neutral pieces) on the sites corresponding to the coordinates B2, D2 and B4. The last line places randomly a component called "Ball" shared by all the players on 29 empty sites.

For all these placement starting rules, Ludii checks if the pieces named in them are defined in the equipment. If they are not, the game will not compile, giving a warning to the game designer of their mistake.

The other common starting rules ludeme is the Set ludeme. Many data can be initiated with this ludeme. Here are some examples

✳ Ludeme 22
```
(set Cost 5 Vertex at:10)
(set Phase 1 Cell at:3)
(set Amount 5000)
(set score P1 50)
(set Team 1 {P1 P3})
(set Hidden at:1 to:P1)
(set Hidden What at:5 level:1 to:P2)
```

The first line sets the cost of the vertex 10 to 5. The second line set the phase of the cell 3 to 1. The third line set the amount of all the players to 5000. The fourth line sets the score of the player 1 to 50. The fifth lines puts the player 1 and 3 in the team 1. The sixth line makes invisible the site 1 to the player 1. The last line makes the piece index of site 5 at the level 1 hidden to the player 2.

Some starting rules exist also to iterate through data of the game to create the initial state. They all start with (forEach ...) and are similar to their equivalents in the Moves ludemes. Here are some examples

✳ Ludeme 23
```
(forEach Player (set Hidden What at:1 to:Player))
(forEach (players Ally of:(next)) (set Hidden What at:1 to:Player))
(forEach Value min:1 max:5 (set Hidden at:10 level:(value) to:P1))
(forEach Site (sites Top) if:(is Odd (site)) (place "Spy1" (site)))
```

The two last starting rules are the Split ludeme and the Deal ludeme. Both are used for cards or dominoes games. The first ludeme can split a deck of cards in two equals decks, the second one deal a certain number of cards/dominoes to each player. However, both can be modified in a future version of Ludii.

4.3 Play Rules

The playing rules of a game describe the legal moves of the mover in the current state. These legal moves are defined in the Play ludeme thanks to Moves ludemes.
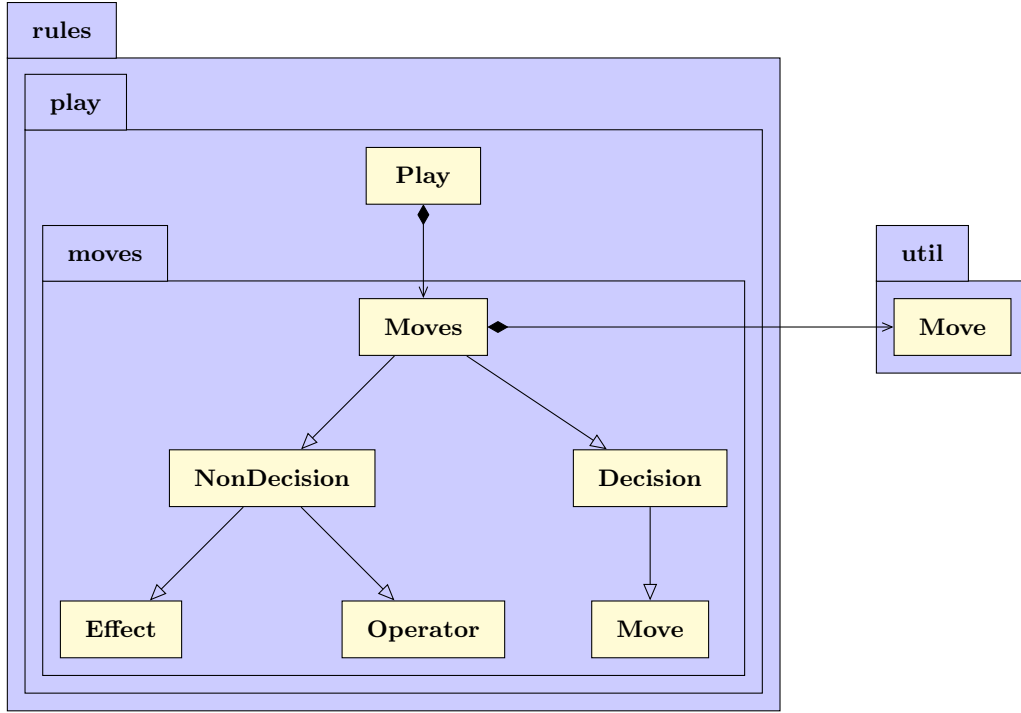
Each time a new state is reached, the (play ...) ludeme evaluates the Moves object stored in it in calling the method Moves eval(Context context). This method returns a Moves object containing a list of util.Move objects (see Section 2.5.1) modelling the list of all the legal moves and an object called Then. The Then object corresponds to the consequences of each legal move. This object contains its own Moves object evaluated and applied after applying the move decided by the mover.

The Moves object can be a decision or not as shown in the UML diagram above. A decision move involves a decision taken by a player. A non decision move is an effect applied because of a decision (see Section 4.3.4) or is an operator (see Section 4.3.3).

4.3.1 Decision Moves

Ludii has only one single decision move called Move. This ludeme modifies internally all the Move objects evaluated thanks to the eval method. Inside each Move object only one action can be a decision. If the Move object is a decision, the data about the first decision action in the list of actions becomes the data of the move. These data are used by Ludii and the user interface to identify a specific move in the list of legal moves when a decision is taken.

For example, the move m described by the list of actions $L = \langle\ ActionRemove(20),$ $ActionMove(10, 20),\ ActionSetScore(1, 10)\ \rangle$ moves a piece from the site 10 to the site 20, removes what is on the site 20 before the move and sets the score of the player 1 to 10. The decision action is the action $ActionMove(10, 20)$. When the user clicks on the site 10 then the site 20, the move m is selected and the list L of actions is applied to modify the state s_i to reach the state s_{i+1}.

Here is the list of the possible decision moves, followed by a short description:

Decision Move	Description
(move ...)	To decide to move from a site to another.
(move Add ...)	To decide to add (a) piece(s) on a site or a set of sites.
(move Claim ...)	To decide to claim a site or a set of sites.
(move Bet ...)	To decide to bet.
(move Hop ...)	To decide to hop.
(move Leap ...)	To decide to leap.
(move Pass ...)	To decide to pass.
(move PlayCard ...)	To decide to play a card.
(move Promote ...)	To decide to promote a piece.
(move Propose ...)	To decide to propose a subject.
(move Remove ...)	Used to remove (a) piece(s) placed on a site or a set of sites.
(move Set Rotation ...)	To decide to modify the rotation state of a piece.
(move Set NextPlayer ...)	To decide to modify the next player to play.
(move Set TrumpSuit ...)	To decide to set the trump suit in a card game.
(move Select ...)	To decide to select sites.
(move Shoot ...)	To decide to shoot.
(move Slide ...)	To decide to slide.
(move Step ...)	To decide to step.
(move Swap Pieces ...)	To decide to swap two pieces placed on two different sites.
(move Swap Players ...)	To decide to swap two players.
(move Vote ...)	To decide to vote for a subject.

4.3.2 Non Decision Moves

The non decision moves ludemes are composed of all the other moves ludemes. Most of them are effects applied because of a decision and a few are operators used to combine or filter moves.

4.3.3 Moves Operator

The Moves operators are used commonly to describe the list of legal moves in generating them in combining many moves or in filtering moves according to some constraints or conditions.

This section briefly describes the logic behind each operator:

Operator	Description
(allCombinations ...)	Generates all combinations in making the cross product between two lists of Move objects.
(and ...)	Combines lists of moves in one list. This ludeme is commonly used in a Then ludeme to apply many consequences.
(append ...)	Appends the list of moves to each move in the list.

(forEach Die ...)	Generating a list of moves in iterating the values of each die (and optionally with combinations of dice).
(forEach Direction ...)	Generating a list of moves in iterating each site reached by each possible direction from a site.
(forEach Group ...)	Generating a list of moves in iterating each group of pieces.
(forEach Piece ...)	Generating a list of moves according to the generators of moves implemented in each component owned by a specific player.
(forEach Player ...)	Generating a list of moves in iterating through the indices of the players with the ludeme (player).
(forEach Site ...)	Generating a list of moves in iterating each site in a region.
(forEach Team ...)	Generating a list of moves in iterating through the teams with the ludeme (team).
(forEach Value ...)	Generating a list of moves in iterating between two integers or on an IntArrayFunction.
(if ...)	Returning a list of moves if a condition is verified else returning (potentially) another list of moves.
(or ...)	Combines lists of moves in one list.

The iterating operators using a (forEach ...) loop are referring to the iterated element by different ludemes.

To iterate through dice, the ludeme (forEach Die ...) uses the ludeme (pips) to get the value (current number of pips) of each die. In the following example, each piece owned by the mover can move to another site in following a track. The number of steps in the track is defined by the number of pips in each die.

⌘ **Ludeme 24**
```
(forEach Die
    (move
        (from (sites Occupied by:Mover))
        (to (trackSite Move steps:(pips)))
    )
)
```

To iterate through directions, the ludeme (forEach Direction ...) modifies the value of the ludeme (to) to each site reached for each direction from a site. In the following example, a piece can move to any empty diagonal adjacent site from its current position.

To iterate through groups of pieces, the ludeme (forEach Group ...) uses the ludeme (sites) to get the region (set of sites) of each group of pieces. In the following example, a score of 1 is added to the mover for each group of pieces bigger than one piece owned by the mover.

To iterate through the pieces, the ludeme (forEach Piece ...) uses the ludeme (from) to iterate each position of the pieces in using the moves defined in the generator associated to the component. In the following example, the pieces named "Ball" owned by the mover are allowed to step to all the empty adjacent sites.

To iterate through a region, the ludeme (forEach Site ...) uses the ludeme (site) to iterate each site in the region. In the following example, all the states of the sites occupied by a piece owned by the mover are set to 1.

Ludeme 28

```
(forEach Site
    (sites Occupied by:Mover)
    (set State at:(site) 1)
)
```

To iterate through teams, the ludeme (forEach Team ...) uses the ludeme (team) to iterate each team. In the following example, the piece index on the site 1 is hidden to each player on a team.

Ludeme 29

```
(forEach Team
    (forEach (team)
        (set Hidden What at:1 of:Player)
    )
)
```

To iterate over a range of integers, the ludeme (forEach Value ...) uses the ludeme (value). In the following example, each piece owned by the mover can move to another site in following a track. The number of steps in the track is between 1 and 6.

Ludeme 30

```
(forEach Value min:1 max:6
    (move
        (from (sites Occupied by:Mover))
        (to (trackSite Move steps:(value)))
    )
)
```

4.3.4 Effect Moves

The effects are moves corresponding to a list of actions to apply as an effect of a decision. These effects can be applied before or after the decision according to their use. All the effects moves ludemes encapsulated in a consequence by the ludeme Then (see Section 4.3) will be applied after the move itself and will refer to the previous state, commonly in using (last From) and (last To) to obtain the site from where began the move to the site where the piece has moved.

For example, in the following example, a piece can step from its current site to any adjacent site occupied by an enemy piece. After that move is applied, the consequence (remove (last To)) is applied, removing the piece just moved. Assuming the move was from the site 5 to the site 10, the list L of actions to apply is $L = \langle\, ActionMove(5, 10),\, ActionRemove(10)\,\rangle$.

```
(move
    Step
    (from)
    (to
        if:(is Enemy (who at:(to)))
    )
    (then
        (remove (last To))
    )
)
```

The other effects which are different than a consequence will be applied before the decision move. For example, in the following example, a piece can step from its current site to any adjacent site occupied by an enemy piece. But before to move, the enemy piece on the site to move is removed because of the ludeme `(apply (remove (to)))`. Assumed the move was from the site 5 to the site 10, the list L of actions to apply is $L = \langle\ ActionRemove(10),$ $ActionMove(5, 10)\ \rangle$.

```
(move
    Step
    (from)
    (to
        if:(is Enemy (who at:(to)))
        (apply
            (remove (to))
        )
    )
)
```

It is important to note that the moves encapsulated inside a (then ...) ludeme are not referring to the current state but to a state modified by the application of the move corresponding to the decision of the mover. That's why, it is needed to use the ludemes (last To) and (last From) inside that ludeme to refer to the 'from' and 'to' sites of the decision just taken and not any variable set during the computation of the moves such as (from), (to), (between), (site) or any iterators. These different iterators will be already reset to their default value when the computation of the moves inside the (then ..) ludeme start.

In the following example, all the occupied sites by a piece owned by the mover can be selected. When a decision is taken to select a site, the piece on that site is flipped. We can see here that the site to flip is referred with (last To) in the (then ...) move contrary to (site) outside of it.

4.3.5 Requirement Moves

The requirement moves define criteria that must be satisfied for a move to be legal. These are typically applied to lists of generated moves to filter out those that do not meet the specified criteria.

This section briefly describes the logic behind each requirement move:

Operator	Description
(avoidStoredState ...)	Filters the legal moves to avoid to reach a specific state.
(do ...)	Applies a sequence of moves in a specified order. Can also filter some moves according to conditions on the state reached after applying that move.
(firstMoveOnTrack ...)	Returns the first legal move on the track.
(max Distance ...)	Filters the moves to keep only the moves allowing the maximum distance on a track in a turn.
(max Captures ...)	Filters a list of moves to keep only the moves doing the maximum possible number of removing actions (captures).
(max Moves ...)	Filters a list of legal moves to keep only the moves allowing the maximum number of moves in a turn.
(priority ...)	Returns the first list of moves with a non-empty set of moves.
(while ...)	Applies a move while a condition is true.

The ludeme (avoidStoredState ...) is twinned with the ludeme (remember State). When (remember State) is called, the current state is stored and when (avoidStoredState ...) encapsulates a Moves ludeme, all moves reaching the state stored in applying it is removed from the list of moves computed.

The ludeme (do ...) can be used in two manners. The first one is to apply a set of Moves before to apply another one, then if the moves corresponding to the next part is not empty, the corresponding moves are returned. In the following example, all the dice are rolled before to compute the moves of each piece.

⚜ **Ludeme 34**
```
(do (roll)
    next:(forEach Piece)
)
```

The second manner is to compute a set of moves, then to filter them according to some conditions on the state reached. On the following example, all the moves of each piece are

computed and all the ones reaching a state where all the pieces of the next player are removed from the list of moves returning by the ludeme (do ...).

> ✖ **Ludeme 35**
> ```
> (do
>     (forEach Piece)
>     ifAfterwards:(!= 0 (count Sites occupied by:Next))
> )
> ```

Move requirements such as (do ...) can be quite powerful when used correctly, but care must be taken as they can have a high performance overhead.

4.3.6 Set Moves

The Set Moves ludeme sets some aspect of the game state:

Operator	Description
(set Count ...)	Sets the count of a site (the number of pieces of the same type on that site).
(set Counter ...)	Sets the counter of the game.
(set Hidden ...)	Makes the site invisible to a player.
(set Hidden Count ...)	Makes the number of pieces on a site hidden to a player.
(set Hidden Rotation ...)	Makes the piece rotation on a site hidden to a player.
(set Hidden State ...)	Makes the state site hidden to a player.
(set Hidden Value ...)	Makes the piece value on a site hidden to a player.
(set Hidden What ...)	Makes the piece index on a site hidden to a player.
(set Hidden Who ...)	Makes the piece owner on a site hidden to a player.
(set NextPlayer ...)	Sets the player to play in the next turn.
(set Pending ...)	Sets the state in pending with specific value(s).
(set Pot ...)	Sets the pot.
(set Rotation ...)	Sets the rotation state of a piece.
(set Score ...)	Sets the score of a player.
(set State ...)	Sets the state of a site.
(set Team ...)	Sets a team of players.
(set TrumpSuit ...)	Sets the trump suit of the card game.
(set Value ...)	Sets the value associated to a player.
(set Var ...)	Sets a variable Var associated to the state.

4.3.7 Take Moves

The (take ...) ludeme is a generic ludeme for any move related to take:

Operator	Description
(take Control ...)	Replaces all the pieces of a player by the equivalent pieces to another player.
(take Domino ...)	Takes a domino from those remaining in the bag.

Concerning the (take Control ...) ludeme, it is important to define in the equipment the same pieces for all the players for it to fully work. The pieces to take control without equivalent are removed.

4.3.8 State Moves

The State Moves ludemes are ludemes modifing the state similarly to the Set Moves.

Operator	Description
(addScore ...)	Add a value to the current score of a player.
(forget Value ...)	Remove one or more value(s) kept in memory by previous state.
(moveAgain)	Set the next player to the mover for him to play again in the next state.
(remember State)	Store the current state in order to avoid to go back to it with the ludeme (avoidStoredState ...).
(remember Value ...)	Store a value to keep in memory for future states.
(swap Players ...)	Swap two players (in switching their name).
(swap Pieces ...)	Swap two pieces placed on two different sites.

4.3.9 Others

Operator	Description
(add ...)	To add (a) piece(s) on a site.
(apply ...)	To apply an effect if a condition is satisfied.
(attract ...)	To attract all the pieces as close as possible to a site in each direction.
(bet ...)	To bet an amount.
(claim ...)	To claim a site by adding a piece of the colour of the owner.
(custodial ...)	To apply an effect to all flanked sites.
(deal ...)	To deal dominoes or cards to the players.
(directionCapture ...)	To capture all the pieces in the same or opposite direction of the last move.
(enclose ...)	To apply an effect to all the pieces enclosed between group(s) of pieces containing the last piece moved or placed.

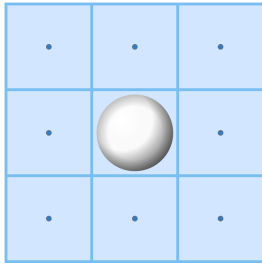|	
(flip ...)	To flip a piece from its current state to the corresponding flipped state defined in the piece.
(fromTo ...)	To move a piece from a site to another site.
(hop ...)	To jump from a site to another in following a radial.
(intervene ...)	To apply an effect to pieces flanking (a) site(s).
(leap ...)	To move to a site resulting from a turtle graphic walk.
(note ...)	To notify a player with a message.
(pass ...)	To pass the turn.
(playCard ...)	To play a card from a fan to the board.
(promote ...)	To promote a piece to another.
(propose ...)	To propose a subject to vote.
(push ...)	To push pieces in a direction.
(remove ...)	To remove (a) piece(s) placed on a site or a set of sites.
(roll ...)	To roll dice.
(satisfy ...)	To return the legal moves of a deduction puzzle.
(select ...)	To select one or two sites (from and to).
(shoot ...)	To add a piece to any site in the line of sight of a piece.
(slide ...)	To move a piece to any site in its line of sight.
(sow ...)	To sow pieces to many sites in following a track.
(step ...)	To step a piece from a site to another connected one.
(surround ...)	To apply an effect to all the sites surrounded.
(then ...)	To encapsulate effects in a consequence moves.
(trigger ...)	To trigger an event.
(vote ...)	To vote for a subject.
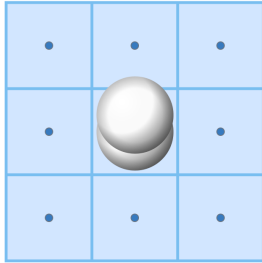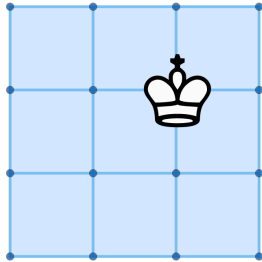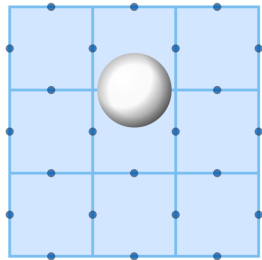
In the following part, more examples and explanations are providing for the most common moves with many possible uses. The blue/red dots corresponds to the legal moves according to these descriptions.

4.3.9.1 Add

The ludeme *Add* can be used to add one or many pieces on different sites and according to different conditions. It is generally used as a decision move, an effect or a consequence. The main parameters of this ludeme are:

- (piece ...) describes the piece to add (Default: (piece (mover))).
- (to ...) describes the site or region to place add the piece.
- count: specifies the number of pieces to add (Default: 1).
- stack: specifies if the piece is added on top of a stack or not.

Here are some examples of how it's used in a 3x3 square board using the cells by default.

Description	Result

A ball can be added to the empty cells.

⌘ Ludeme 36
```
(move Add
    (to
        (sites Empty)
    )
)
```


Two balls can be added to the empty cells.

⌘ Ludeme 37
```
(move Add
    (to (sites Empty))
    count:2 stack:true
)
```


A king can be added to the empty vertices.

⌘ Ludeme 38
```
(move Add
    (piece (id "King" Mover))
    (to Vertex
        (sites Empty Vertex)
    )
)
```


A ball can be added to the empty edges.

⌘ Ludeme 39
```
(move Add
    (to Edge
        (sites Empty Edge)
    )
)
```

A disc can be added only on enemy stack of size 1.

Ludeme 40
```
(move Add
    (to
        (sites Occupied by:Next)
        if:(= 1 (size Stack at:(to)))
    )
    stack:true
)
```
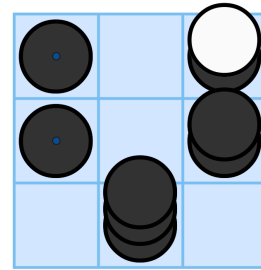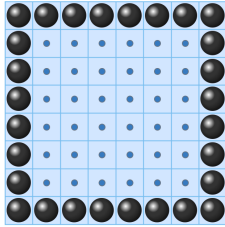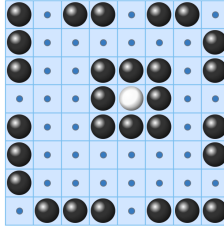

4.3.9.2 Attract

The ludeme *Attract* used to attract all the pieces in different direction from a site (e.g. Feed the Ducks[19]) is generally used as a consequence after moving or placing a piece. The main parameters of this ludeme are:

- (from ...) describes the attracting site (default: (from (last To))).

- The direction to attract (default: Adjacent).

Here are some examples of how it's use in a 8x8 square board using the cells by default:

Description	Before	After
Attract pieces in all directions.		

Ludeme 41
```
(move
    Add
    (to (sites Empty))
    (then
        (attract (from (last To)))
    )
)
```


---

[19]Feed the Ducks: nestorgames.com

Attract pieces in orthogonal directions.

✺ Ludeme 42

```
(move
    Add
    (to (sites Empty))
    (then
        (attract
            (from (last To))
            Orthogonal
        )
    )
)
```
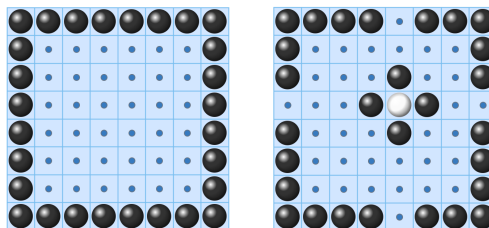

4.3.9.3   Custodial

The ludeme *Custodial* is generally used as a consequence after moving or placing a piece to capture flanked pieces (e.g. Hnefatfl[20]). The main parameters of that ludeme are:

- (from ...) describes the site possibly causing the custodial effect (default: (from (last To))).

- (between ...) describes the data about the flanked site(s). (between) ludeme is used to iterate each potential flanked site. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...). It is also possible to define the maximum number of pieces to flank in each direction.

- (to ...) describes the flanking sites. The (to) ludeme is used to iterate each potential flanking site. The condition on that site is described with the parameter "if:".

Here are some examples of how it's used in a hex board of size 3 using the cells by default:

Description	Before/After

---

[20]Talf games: Wikipedia

After stepping a piece, each flanked enemy is removed.

� Ludeme 43

```
(move Step
    (to if:(is Empty (to)))
    (then
        (custodial
            (from (last To))
            (between
                (max 1)
                if:(is Enemy (who at:(between)))
                (apply (remove (between)))
            )
            (to if:(is Friend (who at:(to))))
        )
    )
)
```
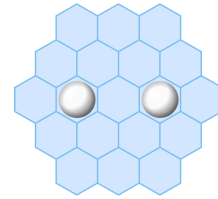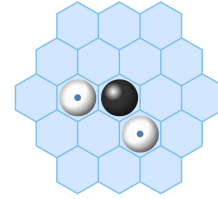
After stepping a piece, any line of flanked enemy pieces is removed.

�֎ Ludeme 44

```
(move Step
    (to if:(is Empty (to)))
    (then
        (custodial
            (from (last To))
            (between
                if:(is Enemy (who at:(between)))
                (apply (remove (between)))
            )
            (to if:(is Friend (who at:(to))))
        )
    )
)
```
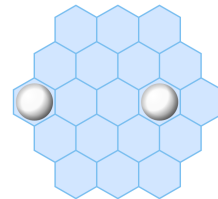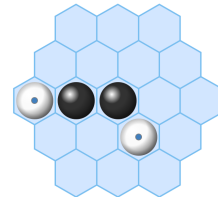
4.3.9.4 DirectionCapture

The ludeme *DirectionCapture* is generally used as a consequence after moving or placing a piece (e.g. Fanorona[21]) to capture all the pieces in the same or opposite direction of the move. The main parameters of that ludeme are:
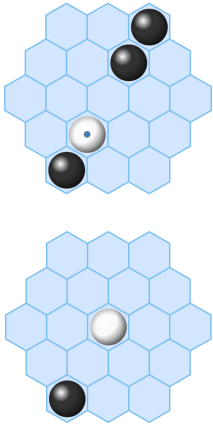
- (from ...) describes the site starting the capture (default: (from (last To))). In case of a capture in the opposite direction that site would have to be (from (last From)).

- (to ...) described the data of the sites. The (to) ludeme is used to iterate each site in the direction, when one site does not satisfy the condition the ludeme stops to check the next sites in that direction. The condition is described with the parameter "if:" and the

---

[21]Fanorona: Fanorona

60

effect to apply described inside the ludeme (apply ...).

Here are some examples of how it's used in a hex board of size 3 using the cells by default:

Description	Before/After
After stepping a piece, all the enemy pieces in the same direction of the move are removed.	


```
Ludeme 45
(move Step
    (to if:(is Empty (to)))
    (then
        (directionCapture
            (from (last To))
            (to
                if:(is Enemy (who at:(to)))
                (apply (remove (to)))
            )
        )
    )
)
```

After stepping a piece, all the enemy pieces in the opposite direction of the move are removed.


```
Ludeme 46
(move Step
    (to if:(is Empty (to)))
    (then
        (directionCapture
            (from (last From))
            (to
                if:(is Enemy (who at:(to)))
                (apply (remove (to)))
            )
            opposite:true
        )
    )
)
```

4.3.9.5 Enclose
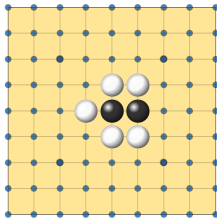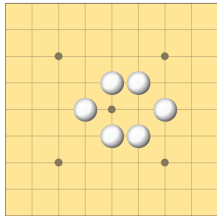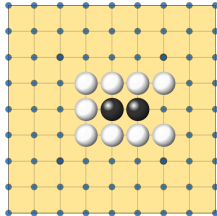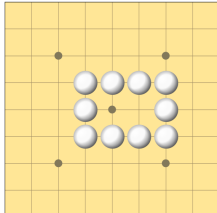
The ludeme *Enclose* is generally used as a consequence after moving or placing a piece (e.g. Go[22]) to capture all the pieces enclosed because of the last move. The main parameters of that ludeme are:

- (from ...) describes the site possibly causing the enclose effect (default: (from (last To))).

---

[22]Go: Wikipedia

- The directions used to enclose pieces (default: Adjacent).

- (between ...) describes the data on the enclosed sites. The (between) ludeme is used to iterate each potential enclosed site. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

Here are some examples of how it's used in a 9x9 go board using the vertices by default:

Description	Before/After
After placing a piece, the enemy pieces enclosed orthogonally are removed.	

Ludeme 47
```
(move Add
    (to (sites Empty))
    (then
        (enclose
            (from (last To))
            Orthogonal
            (between
                if:(is Enemy (who at:(between)))
                (apply (remove (between)))
            )
        )
    )
)
```

After placing a piece, the enemy pieces enclosed adjacently are removed.

Ludeme 48
```
(move Add
    (to (sites Empty))
    (then
        (enclose
            (from (last To))
            (between
                if:(is Enemy (who at:(between)))
                (apply (remove (between)))
            )
        )
    )
)
```

4.3.9.6 Hop
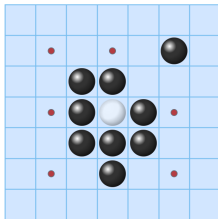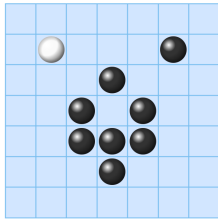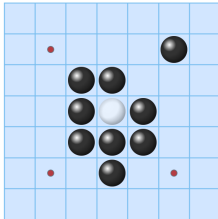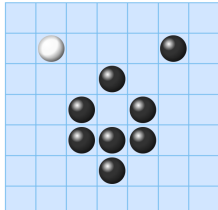
The ludeme *Hop* is used to jump over site(s) (e.g. Draughts[23]) and potentially to capture the pieces placed on the sites jumped. It is generally used in the generator of moves of the pieces.

---

[23]Draughts: Wikipedia

The main parameters of that ludeme are:

- (from ...) describes the origin site of the move (default: (from)).

- The directions allowed to jump (default: Adjacent).

- (between ...) describes the data on the jumped sites. The (between) ludeme is used to iterate each potential jumped site. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...). The maximum number of sites before to jump is described with the parameter "before:", the range of sites to jump is described with a RangeFunction and the maximum number of sites to stop the move after the jumped sites is described with the parameter "after:".

- (to ...) describes the data on the sites to move. The (to) ludeme is used to iterate each potential target site. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

Here are some examples of how it's used in a 7x7 square board using the cells by default:

Description	Before/After

A piece can jump an enemy piece to an empty site to capture it.
 	
 ✿ Ludeme 49
 ```(move Hop```
 ```   (between```
 ```   if:(is Enemy (who at:(between)))```
 ```   (apply (remove (between)))```
 ```   )```
 ```   (to if:(is Empty (to)))```
 ```)```	

A piece can jump a diagonal enemy piece to an empty site to capture it.
 	
 ✿ Ludeme 50
 ```(move Hop```
 ```   Diagonal```
 ```   (between```
 ```   if:(is Enemy (who at:(between)))```
 ```   (apply (remove (between)))```
 ```   )```
 ```   (to if:(is Empty (to)))```
 ```)```	

A piece can jump an orthogonal enemy piece to an empty site to capture it.

✿ Ludeme 51

```
(move Hop
     Orthogonal
     (between
          if:(is Enemy (who at:(between)))
          (apply (remove (between)))
     )
     (to if:(is Empty (to)))
)
```
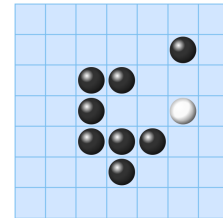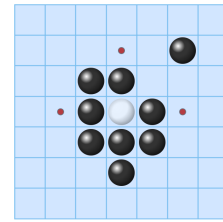

A piece can jump an orthogonal enemy piece to a site occupied by an enemy to capture both enemy pieces.

✿ Ludeme 52

```
(move Hop
     Orthogonal
     (between
          if:(is Enemy (who at:(between)))
          (apply (remove (between)))
     )
     (to
          if:(is Enemy (who at:(to)))
          (apply (remove (to)))
     )
)
```
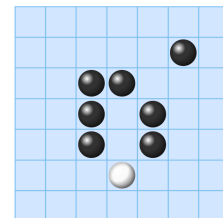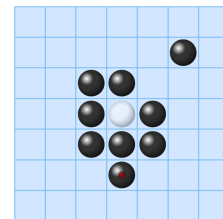

A piece can jump one or two orthogonal enemy pieces to an empty site in capturing the jumped pieces.

✿ Ludeme 53

```
(move Hop
     Orthogonal
     (between
          (range 1 2)
          if:(is Enemy (who at:(between)))
          (apply (remove (between)))
     )
     (to if:(is Empty (to)))
)
```
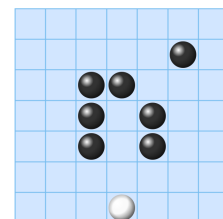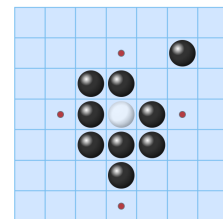
A piece can jump an orthogonal enemy piece to an empty site at a maximum distance of 2 after the jumped sites in capturing the jumped piece.

⚙ Ludeme 54
```
(move Hop
     Orthogonal
     (between
          after:2
          if:(is Enemy (who at:(between)))
          (apply (remove (between)))
     )
     (to if:(is Empty (to)))
)
```
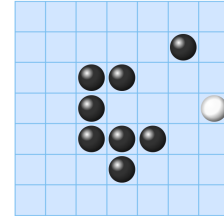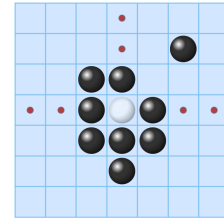
A piece can jump a diagonal enemy piece to an empty site at a maximum distance of 2 before the jumped sites in capturing the jumped pieces.

⚙ Ludeme 55
```
(move Hop
     Diagonal
     (between
          before:2
          if:(is Enemy (who at:(between)))
          (apply (remove (between)))
     )
     (to if:(is Empty (to)))
)
```
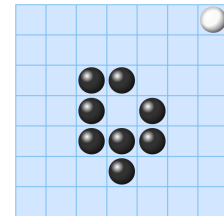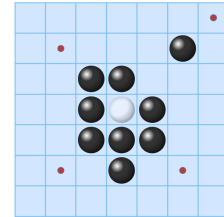
4.3.9.7 Intervene

The ludeme *Intervene* is generally used as a consequence after moving or placing a piece to capture pieces flanking the target site (e.g. Mak-yek[24]). The main parameters of that ludeme are:

- (from ...) describes the possible flanking site (default: (from (last To))).

- (to ...) describes the data on the flanking sites. The (to) ludeme is used to iterate each potential flanking site. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

Here is a example of how it's used in a hex board of size 3 using the cells by default:

Description	Before/After

---

[24]Mak-yek: Wikipedia

After stepping a piece, the enemy pieces flanking the moved piece are removed.

> **⌘ Ludeme 56**
>
> ```
> (move Step
>     (to if:(is Empty (to)))
>     (then
>         (intervene
>             (from (last To))
>             (to
>                 if:(is Enemy (who at:(to)))
>                 (apply (remove (to)))
>             )
>         )
>     )
> )
> ```
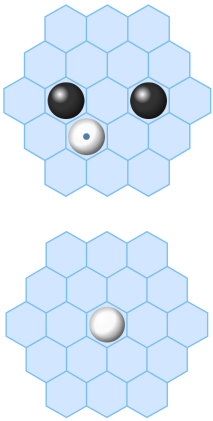
4.3.9.8 Leap

The ludeme *Leap* is used to jump sites by following a Turtle graphics walk (see Wikipedia) like the knight in Chess[25] to move or capture pieces by replacement. It is generally used in the generator of moves of the pieces. The main parameters of that ludeme are:

- (from ...) describes the origin site of the move (default: (from)).

- The walk(s) to follow described between braces { and }.

- (to ...) describes the date on the sites to move. The (to) ludeme is used to iterate each potential site to move. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

By default, all the orthogonal rotations of the walk will be used to compute the legal moves. However, that's possible to filter them in keeping only the forwards moves (according to the facing direction of the piece) thanks to the parameter "forward:true" and that's also possible to compute the legals moves with not all the rotations but in using only the primary direction (the north for a square board) with the parameter "rotations:false".

Here are some examples of how it's used in a square board of size 7 using the cells by default:

Description	Before/After

---

[25]Chess: Wikipedia

A knight piece can leap as a chess knight to move to an empty site or to capture an enemy piece.

⌘ Ludeme 57

```
(move Leap
    { {F F R F} {F F L F} }
    (to
        if:(not (is Friend (who at:(to))))
        (apply
            (if (is Enemy (who at:(to)))
                (remove (to) )
            )
        )
    )
)
```
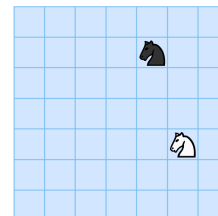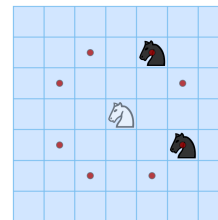

A knight piece facing to the north can leap as a large L only in the forward directions to move to an empty site or to capture an enemy piece.

⌘ Ludeme 58

```
(move Leap
    { {F F F R F} {F F F L F} }
    forward:true
    (to
        if:(not (is Friend (who at:(to))))
        (apply
            (if (is Enemy (who at:(to)))
                (remove (to) )
            )
        )
    )
)
```
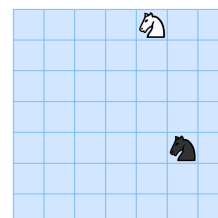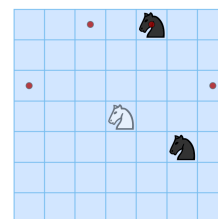
A knight piece can leap with a specific walk (not all the rotations) to move to an empty site or to capture an enemy piece.

✵ Ludeme 59

```
(move Leap
    { {F F F R F} }
    rotations:false
    (to
        if:(not (is Friend (who at:(to))))
        (apply
            (if (is Enemy (who at:(to)))
                (remove (to) )
            )
        )
    )
)
```
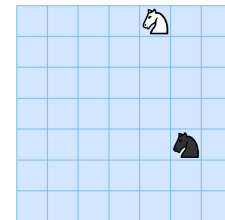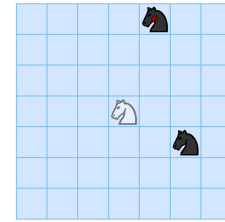
4.3.9.9 Promote

The ludeme *Promote* uses to convert a piece to another can be used as a consequence after moving or placing a piece or as a decision move (e.g. Chess[26]). The main parameters of that ludeme are:

- The site where is the piece to promote.
- The list of possible promotion piece types described in a (piece ...) ludeme.
- The owner of these piece types.

Here is a example of how it's used in a square board of size 3 using the cells by default:

Description	Before/After

A Pawn piece can promote to a queen, a knight, a bishop or a rook of the current player. **✵ Ludeme 60**
 	
```(move Promote```
```   (where (id "Pawn" Mover))```
```   (piece {"Queen" "Knight" "Bishop" "Rook"})```
```   Mover```
```)```	

---
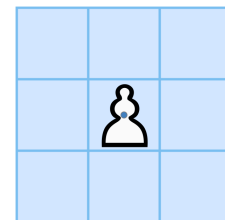
[26]Chess: Wikipedia

4.3.9.10 Push

The ludeme *Push* is generally used as a consequence to push a set of pieces in a specific direction. The main parameters of that ludeme are:

- (from ...) describes the origin site of the move (default (last To)).

- The direction to push.

Here is a example of how it's used in a square board of size 5 using the cells by default:

Description	Before/After
A King piece is selected and pushes all the pieces one step to the north.	

Ludeme 61
```
(move
    Select
    (from
        (sites Occupied by:Mover
            component:"King"
        )
    )
    (then
        (push (from (last To)) N)
    )
)
```

4.3.9.11 Remove

The ludeme *Remove* used to remove one or many piece(s) from the board can be used as an effect or a consequence like in different examples in that document, but can also be used as a decision move.

The main parameters of that ludeme are:

- The site(s) to remove the piece(s) on.

- count: describes the number of pieces to remove.

Here is a example of how it's used in a square board of size 5 using the cells by default:

Description	Before/After

Any piece owned by the current player can be removed from the board.
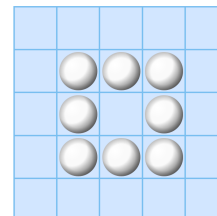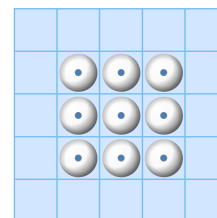
4.3.9.12 Shoot

The ludeme *Shoot* is used to place a piece in any site in the line of sight of another piece (e.g. Amazons[27]). It is generally used has a decision. The main parameters of that ludeme are:

- (piece ...) describes the piece to shoot.
- (from ...) describes the origin site of the move (default (last To)).
- The directions allowed to shoot (default: Adjacent).
- (to ...) describes the date of the site to shoot. The (to) ludeme is used to iterate each potential target site. The condition is described with the parameter "if:". In Each direction, the sites are checked, if the condition is false, the piece can not be shoot in that site and farther in the same direction (default (to if:(is Empty (to)))).

Here are some examples of how it's used in a square board of size 5 using the cells by default:

Description	Before/After

---

[27]Amazons: Wikipedia

A ball is shot from the piece placed in the centre to any sites in any adjacent direction.

✿ Ludeme 63

```
(move Shoot
    (piece "Ball")
    (from (centrePoint))
)
```
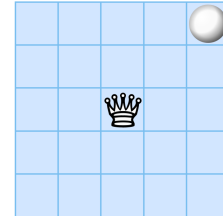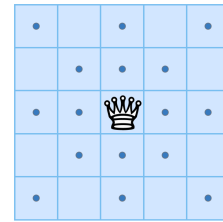
A ball is shot from the piece placed in the centre to any sites in any orthogonal direction.

✿ Ludeme 64

```
(move Shoot
    (piece "Ball")
    (from (centrePoint))
    Orthogonal
    (to if:(is Empty (to)))
)
```
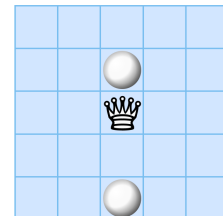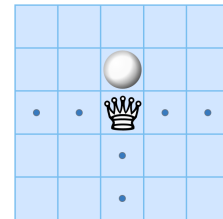
4.3.9.13   Slide

The ludeme *Slide* is used to move a piece to any site in different directions and potentially to capture a piece by replacement (e.g. Chess[28]). We generally use it in the generator of moves of a component.
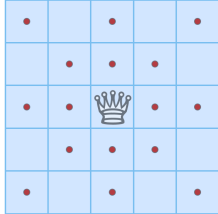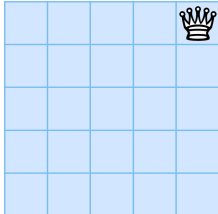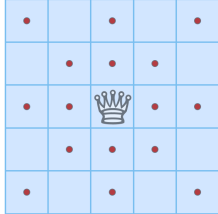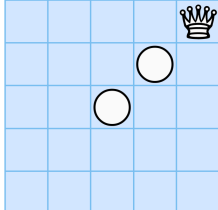
The main parameters of that ludeme are:

- (from ...) describes the origin site of the move (default (from)).

- The directions to slide (default: Adjacent).

- (between ...) describes the data about the sites between the (from) site and the (to) site. The ludeme (between) is used to iterate these sites. The condition is described with the parameter "if:". In Each direction, the sites are checked, if the condition is false, the piece can not be slide in that site and farther in the same direction. The moving piece can trail another piece in each site, that can be described with the parameter "trail:".

---

[28]Chess: Wikipedia

- (to ...) describes the data about the sites to stop to slide. The (to) ludeme is used to iterate these sites. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

Here are some examples of how it's used in a square board of size 5 using the cells by default:

Description	Before/After
A queen piece can slide in each empty site in all the directions. **Ludeme 65** `(move Slide)`	

A queen piece can slide in each empty site in all the directions and trail a "Disc" piece.
 	
Ludeme 66
`(move Slide`
`    (between`
`        trail:(id "Disc")`
`    )`
`)`	

A rook piece can slide in each empty site in all the orthogonal directions and possibly capture an enemy piece.

⌘ Ludeme 67

```
(move Slide
    Orthogonal
    (to if:(is Enemy (who at:(to)))
        (apply (remove (to)))
    )
)
```
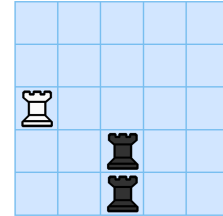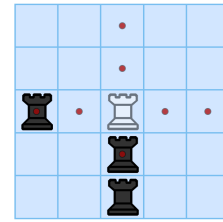
4.3.9.14 Sow

The ludeme *Sow* is used has a consequence after selected a site to sow a certain number of pieces in following a track (e.g. mancala games[29]). The main parameters of that ludeme are:

- count: describes the number of pieces to sow (default: (count at:(last To))).

- numPerHole: describes the number of pieces to sow in each hole (default: 1).

- if: describes the condition on the last site of the sowing to apply an effect on it. That site is described by the ludeme (to).

- sowEffect: describes the effect to apply to each hole sowed.

- apply: describes the effect to apply to the last site of the sowing. That site is described by the ludeme (to).

- includeSelf: if true, the sowing includes the origin site in the sowing (default: true).

- origin: if true, a piece is sowing to the origin site before to begin the sowing (default: false).

- skipIf: describes a condition to skip to sow to a site. That site is described by the ludeme (to).

- backtracking: if true and if the condition to apply the effect in the last site is true, the condition and the effect can also be applied to the previous sites in the sowing until the condition is false (default: false).

- forward: if true and if the condition to apply the effect in the last site is true, the condition and the effect can also be applied to the next sites in the sowing until the condition is false (default: false).

Here is a example of how it's used in a mancala board of 2 rows and 6 columns with stores:

Description	Before/After

---

[29]mancala games: Wikipedia

Any site in the track with at least a piece can be selected then the pieces in it are sowing in following a counter clockwise direction. If after sowing the site has 4 pieces, the pieces are captured to the site mapped to the mover.

❖ Ludeme 68

```
(equipment {
    (mancalaBoard 2 6
        (track "Track" "1,E,N,W"
            loop:true
        )
    )
    (piece "Seed" Shared)
    (map {
        (pair P1 FirstSite)
        (pair P2 LastSite)
    })
})

(move Select
    (from
        (sites Track)
        if:(> (count at:(from)) 0)
    )
    (then
        (sow
            if:(= (count at:(to)) 4)
            apply:(fromTo
                (from (to))
                (to (mapEntry (mover)))
                count:(count at:(to))
            )
        )
    )
)
```
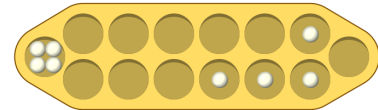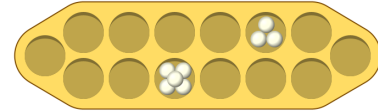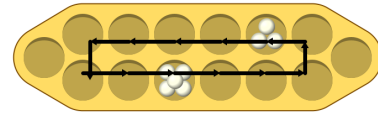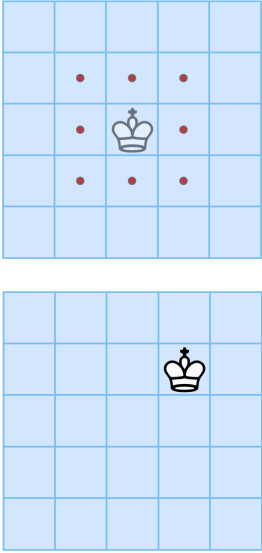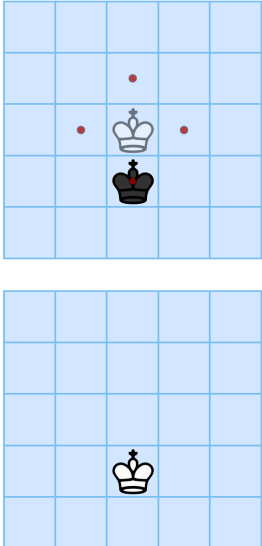
4.3.9.15 Step

The ludeme *Step* can be used to step a piece from a site to another connected site like the king in Chess[30] to potentially capture a piece by replacement. It is generally used in the generator of the pieces. The main parameters of that ludeme are:

- (from ...) describes the origin site (default: (from)).

- The directions allowed to step (default: Adjacent).

- (to ...) describes the data about the sites to step. The (to) ludeme is used to iterate each site to move. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

---

[30]Chess: Wikipedia

Here are some examples of how it's used in a square board of size 5 using the cells by default:
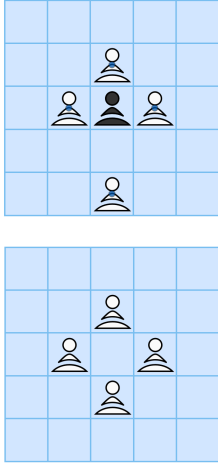
Description	Before/After

A king piece can step to an adjacent empty site or to a site occupied by an enemy to capture it.
 	
 ⌘ Ludeme 69
 ```(move Step```
 ```   (to if:(is Empty (to)))```
 ```)```	

A king piece can step to an orthogonal empty site or to a site occupied by an enemy to capture it.
 	
 ⌘ Ludeme 70
 ```(move Step```
 ```   Orthogonal```
 ```   (to```
 ```   if:(not (is Friend (who at:(to))))```
 ```   (apply```
 ```   (if (is Enemy (who at:(to)))```
 ```   (remove (to))```
 ```   )```
 ```   )```
 ```   )```
 ```)```	

4.3.9.16 Surround

The ludeme *Surround* is generally used as a consequence to apply an effect on pieces surrounding after placing or moving a piece (e.g. Hnefatfl[31]). The main parameters of that ludeme are:

- (from ...) describes the site possibly causing the surrounding of sites (default: (from)).

- The directions used to surround (default: Adjacent).

- (between ...) describes the data about the potential surrounded sites. The (between) ludeme is used to iterate them. The condition is described with the parameter "if:" and the effect to apply described inside the ludeme (apply ...).

- (to ...) describes the surrounding sites. The (to) ludeme is used to iterate them. The condition is described with the parameter "if:".

Here is a example of how it's used in a square board of size 5 using the cells by default:

Description	Before/After
A thrall piece can step to an empty site, then if any enemy piece orthogonally surrounded by friends pieces is removed.	

Ludeme 71

```
(move Step
    (to if:(is Empty (to)))
    (then
        (surround
            (from (last To))
            Orthogonal
            (between
                if:(is Enemy (who at:(between)))
                (apply (remove (between)))
            )
            (to
                if:(is Friend (who at:(to)))
            )
        )
    )
)
```

4.4 End Rules

The ending rules describe when and how play can terminate for one or more of the players. In Ludii, the ending rules start by condition(s) encapsulated in an (if ...) ludeme, followed by the description of the outcome of the player(s). In games with 2 players or less, an (end ...) rule

---

[31]Talf games: Wikipedia

ludeme describes the end of the game, but for games with more players, the game can continue if play did not yet terminate for at least 2 of the players.

All the conditions are described with BooleanFunctions (see Section 5.1). The conditions can be nested within other (if ...).

Two ways are possible to describe the outcomes of the players:

- (result ...): The result ludeme has 2 parameters:

  - A roletype describing the player or the team assigned to that ending state.
  - The ResultType, which can be Win (the player got the best available rank), Loss (the player got the worst available rank), Draw (all the remaining players get the middle of the available ranks).

- (byScore ...): To end the game and order the players according to their scores. Their final scores can be set in that ludeme too.

The following example gives the best available rank to the mover if a line of size 3 minimum is detected with the pieces of the mover.

✤ Ludeme 72

```
(end
    (if
        (is Line 3)
        (result Mover Win)
    )
)
```

The following example if the player is P1, to get the best available rank it needs to make a line of size 3 minimum else it has to make a line of size 4 minimum.

✤ Ludeme 73

```
(end
    (if (is Mover P1)
        (if
            (is Line 3)
            (result P1 Win)
        )
        (if
            (is Line 4)
            (result P2 Win)
        )
    )
)
```

The following example ends the game if all the player just passed in ordering them according to their final score. This score is set for each player in being their current score added to the size of their territory. The ludeme (byScore) also has a misere version.

> **⚜ Ludeme 74**
> ```
> (end
>     (if (all Passed)
>         (byScore {
>             (score P1 (+ (score P1) (size Territory P1)))
>             (score P2 (+ (score P2) (size Territory P2)))
>         })
>     )
> )
> ```

It is also possible to specify end results that trigger simultaneously for multiple, but not necessarily all, players. The following example causes all players who become blocked to simultaneously lose as soon as they become blocked, and causes a player to win when they complete a connected chain of pieces between their regions.

> **⚜ Ludeme 75**
> ```
> (end {
>     (forEach NonMover if:(is Blocked Player) (result Player Loss))
>     (if (is Connected Mover) (result Mover Win))
> })
> ```

4.4.1 Player Rankings

In every n-player game, there are n ranks to be distributed among the players; ranks 1, 2, ..., n. Low numbers are more desirable than high numbers (i.e., rank 1 is always the best rank).

Whenever a player achieves a "Win" result, that player is assigned the next best rank that has not already been assigned. For instance, in a 2-player game where neither player has achieved an outcome yet, this will normally yield rank 1 for the winner. In a game with 6 players, for example, a "Win" result would yield a rank of 3 if two other players have already previously obtained "Win" results (and hence already claimed ranks 1 and 2).

Similarly, whenever a player achieves a "Loss" result, that player is assigned the next worst rank that has not already been assigned. In a typical 2-player game situation, that would simply be rank 2. In a situation in a 5-player game, where 2 other players have already previously lost (and hence claimed ranks 5 and 4), this would be rank 3.

A "Draw" result always terminates the game, and assigns the average of all remaining, unassigned ranks to all players that are still active. In the simple case of a 2-player game, a "Draw" result means that both players obtain a rank of 1.5. Such non-integer ranks are unusual to see for humans, but convenient for AI implementations because it leads to a result that is naturally in between the the "Win" and "Loss" outcomes, and therefore also leads to correctly ordered preferences for different outcomes. In a 6-player game where one player has already lost, and two players have already won, the remaining unassigned ranks would be 2, 3, and 4. A "Draw" result in such a situation would yield a shared rank of $\frac{2+3+4}{3} = 3$ for each of the three remaining players.

When a "Win" (or "Loss") result triggers for $n > 1$ different players simultaneously, the average of the next n best (or worst) ranks available is assigned to those n players, and those n ranks are counted as already having been assigned. For example, suppose that two players win simultaneously in a 4-player game, with all ranks still left unassigned. The best two ranks available are 1 and 2, which means that these two players obtain a shared rank of $\frac{1+2}{2} = 1.5$.

If at a later stage a third player also wins, ranks 1 and 2 both count as already having been assigned, which means that this third player achieves a rank of 3 (and finally the last remaining player defaulting to rank 4).

<div align="right">

5

</div>

<div align="right">

Functions

</div>

Except the moves, each ludeme corresponds to a particular type of function.

5.1 BooleanFunction

The Boolean functions return true or false. Except the math Boolean functions, all of them are starting by one of these keywords: all, can, is, no, was.

In this section, descriptions of the Boolean functions not specifically dedicated to deduction puzzles appear before the ones designed specifically for them.

5.1.1 Math

The mathematics Boolean functions are all the classical mathematics operators returning a Boolean:

Boolean	Description
(= ...)	Returns true if all the Integer functions return the same value.
(!= ...)	Returns true if all the Integer functions return a different value.
(< ...)	Returns true if the value returning by the first parameter is smaller than the value returning by the second parameter.
(<= ...)	Returns true if the value returning by the first parameter is smaller or equal to the value returning by the second parameter.
(> ...)	Returns true if the value returning by the first parameter is greater than the value returning by the second parameter.
(>= ...)	Returns true if the value returning by the first parameter is greater or equal to the value returning by the second parameter.
(and ...)	Returns true if each function described in the ludeme returns true.

(if ...)	Returns the result of a Boolean function among two according to a condition.
(not ...)	Returns true if the Boolean ludeme described in the ludeme returns false.
(or ...)	Returns true if one of the functions described in the ludeme returns true.
(toBool ...)	Returns true if the result of the function in entry is 0.
(xor ...)	Returns the Boolean result of a xor operation between two values.

5.1.2   All

The "all" Boolean ludeme returns true if many aspects of a game satisfy a condition:

Boolean	Description
(all DiceEqual)	Returns True if all the dice have the same values.
(all DiceUsed)	Returns True if all the dice were used by previous moves in the same turn.
(all Passed)	Returns True if all the players pass in their last turn.
(all Sites ...)	Returns True if all the sites of a region satisfy the condition.
(all Values ...)	Returns True if all the values of an array satisfy the condition.

With the ludeme (all Sites ...), each site is the region is iterated by the ludeme (site). In the following example, the ludeme (all Sites ...) returns true if all the sites occupied by the next player are not currently in a line of size 3 or more.

> **⌘ Ludeme 76**
> ```
> (all Sites
>     (sites Occupied by:Next)
>     if:(not
>         (is Line 3 through:(site))
>     )
> )
> ```

5.1.3   Can

The "can" Boolean ludeme is composed in the current version of a single ludeme:

Boolean	Description
(can Move ...)	Returns True if one of the moves described as a parameter of the ludeme is legal.

5.1.4   Is

The "is" ludeme is the most common Boolean ludeme and can be used to check many aspects of a game:

Boolean	Description
(is Active ...)	Returns True if a specific player did not win or lose.
(is AnyDie ...)	Returns True if any die is equal to a specific value.
(is Blocked ...)	Returns True whether regions cannot possibly be connected by a player.
(is CaterpillarTree ...)	Returns True if the graph is the largest caterpillar tree.
(is Connected ...)	Returns True when regions are connected by a group of pieces.
(is Crossing ...)	Returns True if two edges are crossing.
(is Cycle ...)	Returns True if the game is currently in a cycle of state (a succession of states repeated at least three times).
(is Decided ...)	Returns True if a subject is voted.
(is Empty ...)	Returns True if a site is empty.
(is Enemy ...)	Returns True if a player is an enemy to the mover.
(is Even ...)	Returns True if a value is even.
(is Flat ...)	Returns True if all the pieces have pieces under them or are in the ground (for 3D games).
(is Friend ...)	Returns True if a player is a friend to the mover.
(is Full ...)	Returns True if no site is empty.
(is Hidden ...)	Returns True if the site is invisible to a player
(is Hidden Count ...)	Returns True if the number of pieces on a site is hidden to a player
(is Hidden Rotation ...)	Returns True if piece rotation on a site is hidden to a player
(is Hidden State ...)	Returns True if the state site is hidden to a player
(is Hidden Value ...)	Returns True if the piece value is hidden to a player
(is Hidden What ...)	Returns True if the piece index is hidden to a player
(is Hidden Who ...)	Returns True if the piece owner is hidden to a player
(is In ...)	Returns True if a site belongs to a region.
(is LastFrom ...)	Returns True if the graph element type of the 'from' location of the last move is the expected one.
(is LastTo ...)	Returns True if the graph element type of the 'to' location of the last move is the expected one.
(is Line ...)	Returns True if a line of a specific size is detected.
(is Loop ...)	Returns True if a loop of pieces with at least a site in the middle is detected.
(is Mover ...)	Returns True if a player is the mover.
(is Next ...)	Returns True if a player is the next player to play.
(is Occupied ...)	Returns True if a site is not empty.
(is Odd ...)	Returns True if a value is odd.

(is Pattern ...)	Returns True if the sites following a specific pattern satisfied the conditions.
(is Path ...)	Returns True if a path or a cycle of a specific size with pieces owned by a specific player is detected.
(is Pending ...)	Returns True if the state is in pending (due to the use of (set Pending ...)).
(is PipsMatch ...)	Returns True if the pips of all the dominoes match to their neighbours.
(is Proposed ...)	Returns True if a subject is proposed for voting.
(is Prev ...)	Returns True if a player is the previous player who played.
(is RegularGraph ...)	Returns True if the induced graph (according to the parameters of the ludeme) is regular.
(is Related ...)	Returns True if two sites are related to each other (e.g. Orthogonal or Diagonal).
(is Repeat ...)	Returns True if the current state was previously already reached.
(is SidesMatch ...)	Returns True if the terminus of a tile matches with its neighbours (e.g. Trax).
(is SpanningTree ...)	Returns True if the graph is a spanning tree.
(is Target ...)	Returns True if a set of sites is in a specific configurations of pieces.
(is Threatened ...)	Returns True if a location can be reached by an enemy piece.
(is Tree ...)	Returns True if the graph is a tree.
(is TreeCenter ...)	Returns True if the last site selected is the centre of the tree.
(is Trigger ...)	Returns True if a specific event for a specific player was triggered by the ludeme (trigger ...).
(is Visited ...)	Returns True if a site was already used in the previous moves of the same turn by a player.
(is Within ...)	Returns True if a specific piece is currently placed on at least one site of a region.

In the following part, more examples and explanations are providing for some BooleanFunctions related to space constraints.

5.1.4.1 is Connected

The ludeme (is Connected ...) is used to detect a sequence of pieces connected regions (e.g. Hex[32]).
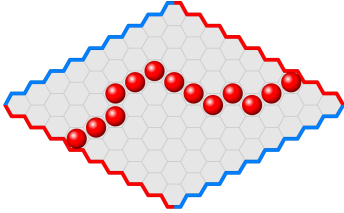
The main parameters of that ludeme are:

- The number of regions to connect (default: All).

- at: describes the site through the connection has to be checked (default: (last To)).

- The directions allowed to connect regions.

---

[32]Hex: Wikipedia

- The disjointed regions to connect (default: the regions associeted with the mover).
- The RoleType of the owner of the pieces making the connection.

Here is a example of how it's used.

Description	Satisfying
A connection of the regions associated with the mover has to be made.	

Ludeme 77
```
(equipment {
    ...
    (regions P1 { (sites Side NE)
        (sites Side SW) } )
    (regions P2 { (sites Side NW)
        (sites Side SE) } )
})

(is Connected Mover)
```

5.1.4.2   is Line

The ludeme (is Line ...) is used to detect a sequence of pieces places in a line pattern (e.g. Gomoku[33]). The main parameters of that ludeme are:

- The length of the line.
- The directions allowed to detect a line (default: Adjacent).
- through: describes the site through has to pass the line (default: (last To)).
- throughAny: describes the region of sites through at least one the line has to pass.
- The RoleType of the owner of the pieces making the line.
- what: describes the index of the component making the line.
- whats: describes the indices of the components making the line.
- exact: if true, line has to have exactly the right length, if false the line can be longer.
- if: describes the condition on the pieces making the line. The ludeme (to) is used to iterate the potential sites in the line.
- byLevel: if true, the line can also be made between different levels of stack of pieces.

Here are some examples of how it's used.

Description	Satisfying

---

[33]Gomoku: Wikipedia

A line of minimum size 3 has to be made.

> ⌘ **Ludeme 78**
> (is Line 3)
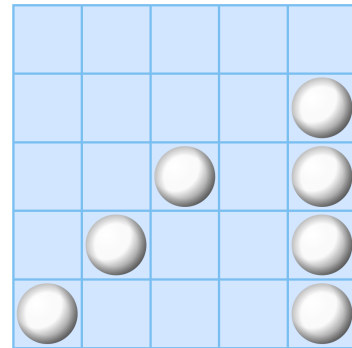
A line of exactly size 3 has to be made orthogonally.

> ⌘ **Ludeme 79**
> (is Line 3 Orthogonal exact:3)
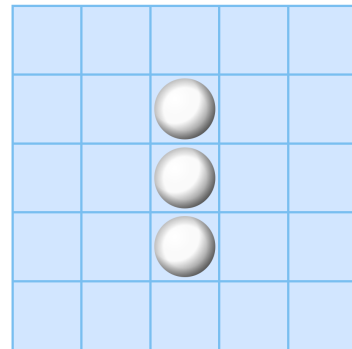
A line of minimum size 3 has to be made with any pieces owned by P1.

> ⌘ **Ludeme 80**
> (is Line 3 P1)
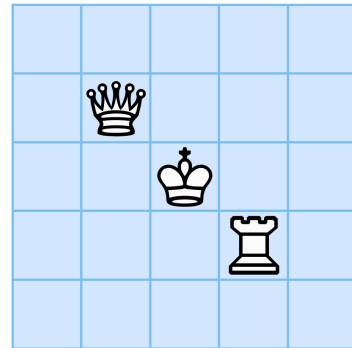
5.1.4.3 is Loop
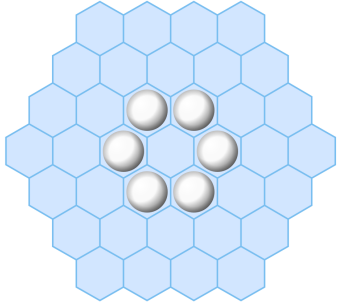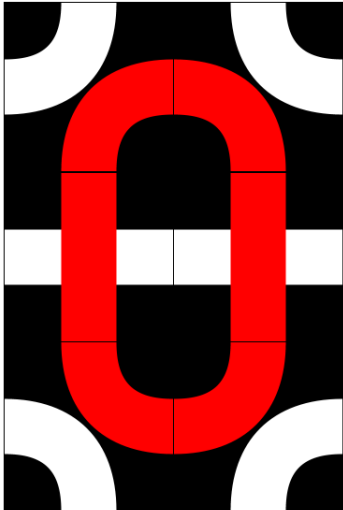
The ludeme (is Loop ...) is used to detect a sequence of pieces looping thought at least a site (e.g. Havannah[34]). The main parameters of that ludeme are:

- The RoleType of the owner of the pieces enclosed by the loop (default: Empty sites).
- The directions allowed to detect the loop (default: Adjacent).
- The indice of the piece used to make the loop (default: (mover)).
- The site through has to pass the loop (default: (last To)).

---

[34]Havannah: Wikipedia

- path: if true, the loop can be detected through tiles (e.g. Trax[35]).

Here are some examples of how it's used.

Description	Satisfying
A loop around empty sites has to be made. **❖ Ludeme 81** `( is Loop )`	
A loop using path of tiles pieces has to be made. **❖ Ludeme 82** `( is Loop path :true )`	
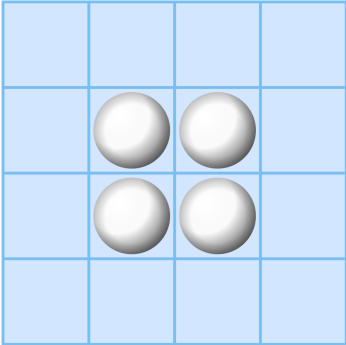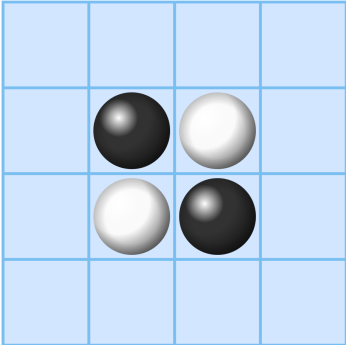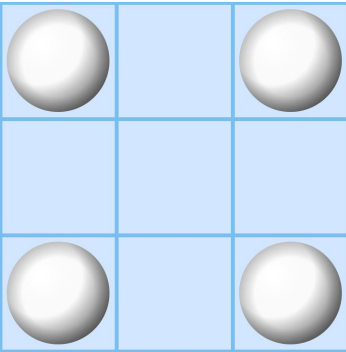
5.1.4.4   is Pattern

The ludeme (is Pattern ...) is used to detect pieces in a specific pattern. The main parameters of that ludeme are:

- The walk describing the pattern.

- from: describes the site through has to pass the pattern (default (last To)).

- what: describes the indice of the piece used to make the pattern (default: (mover)).

- whats: describes the indices of the piece(s) used alternatively to make the pattern.

Here are some examples of how it's used.

---

[35]Trax:Wikipedia

86

Description	Satisfying

A square pattern has to be made.

⌘ Ludeme 83
```
(is Pattern {F R F R F R F})
```

A square pattern has to be made with alternately a piece of the mover and a piece of the next player.

⌘ Ludeme 84
```
(is Pattern {F R F R F R F}
    whats:{(mover) (next)}
)
```

A square pattern of four pieces owned by the mover in a 3x3 square with an empty site between each piece. In this example, the value "0" means an empty site and "(mover)" corresponds to the piece matching the mover index.

⌘ Ludeme 85
```
(is Pattern {F F R F F R F F R F F}
    whats:{(mover) 0 (mover) 0
           (mover) 0 (mover) 0
          }
)
```

5.1.5 No

The "no" Boolean ludeme is composed in the current version of a single ludeme:

Boolean	Description
(no Moves ...)	Returns True if a specific player has no legal moves in this state.
(no Pieces ...)	Returns True if a specific piece type (or all piece types) is not placed.

5.1.6 Was

The "Was" Boolean ludeme is composed in the current version of a single ludeme and refers to a previous state.

Boolean	Description
(was Pass)	Returns True if the last move was a pass.

5.1.7 Deduction Puzzle

Some Boolean functions are designed specially for deductions puzzles. As described in the Section 2.4.3, the deduction puzzles have a special state in Ludii and consequently using any of these Boolean functions in the description of a game will convert the game to a such state.

Boolean	Description
(all Different)	Returns True if all variables are set and have different values.
(is Count ...)	Returns True if the number of values set to variables and equal to a specific value is equal to a specific number.
(is Unique ...)	Returns True if a value is set only one time to variables.
(is Sum ...)	Returns True if the sum of all the values set to variables is equal to a value.
(is Solved ...)	Returns True if all the variables are set and all the constraints of the puzzle are satisfied.

5.2 IntFunction

The Int functions return an integer value. For each of these functions, if nothing is specified in the documentation related to the function, the value UNDEFINED (-1) is returning if the result can not be computed.

5.2.1 Math

The mathematics Int functions are all the classical mathematics operators returning an integer:

IntFunction	Description
(+ ...)	Returns the sum of all the Int functions in parameters of that ludeme.
(- ...)	Returns the result of the subtraction of an Int function by another.
(* ...)	Returns the product of all the Int functions in parameters of that ludeme.
(\ ...)	Returns the (integer) division of an Int function by another.
(% ...)	Returns the modulo of an Int function by another.
(∧ ...)	Returns the result of an Int function to the power of the result of another Int function.
(abs ...)	Returns the absolute value of the result of an Int function.

(if ...)	Returns the result of an Int function among two according to a condition.
(max ...)	Returns the maximal value returning by a set of Int functions.
(min ...)	Returns the minimal value returning by a set of Int functions.
(toInt ...)	Converts a Float function or a Boolean Function to an Int function..

5.2.2 Board

The board Int functions return an integer related to the board:

IntFunction	Description
(ahead ...)	Returns the index of the site at a specific distance in a specific direction from a site.
(centrePoint ...)	Returns the index of the site in the centre of the board. If the centre is many sites, the minimum index between them.
(column of:...)	Returns the index of the column of a site.
(coord ...)	Returns the index of the site corresponding to a coordinate on the board.
(cost ...)	Returns the cost/weight of a graph element.
(handSite ...)	Returns the index of a site in the hand of a player.
(id ...)	Returns the index of a player or of a component.
(layer of:...)	Returns the index of the layer of a site.
(mapEntry ...)	Returns the integer value mapping to the entry.
(phase of:...)	Returns the phase index of a site.
(regionSite ...)	Returns the index of a specific site in a region.
(row of:...)	Returns the index of the row of a site.
(where ...)	Returns the minimal index of a site where is placed a specific piece.

5.2.3 Card

The card Int functions return an integer related to card aspect. However, these ludemes are still in progress and can change in the future:

IntFunction	Description
(card Rank ...)	Returns the index of the rank of a card.
(card Suit ...)	Returns the index of the suit of a card.
(card TrumpRank ...)	Returns the index of the trump rank of a card.
(card TrumpSuit ...)	Returns the index of the suit, which is the trump.
(card TrumpValue ...)	Returns the trump value of a card.

5.2.4 Count

The count Int ludeme is used to count a property within the game such as the number of players, components, sites, turns, groups, etc.

Here is the list of all of them:

IntFunction	Description
(count ...)	Returns the number of pieces in a site (if they are not in a stack).
(count Active)	Returns the number of active players.
(count Adjacent ...)	Returns the number of adjacent sites of a site.
(count Cells)	Returns the number of cells of the board.
(count Columns)	Returns the number of columns of the board.
(count Diagonal ...)	Returns the number of diagonal sites of a site.
(count Edges)	Returns the number of edges of the board.
(count Groups)	Returns the total number of groups of pieces owned by a player.
(count Liberties)	Returns the number of liberties (empty sites) of a group of pieces starting from a site.
(count Moves)	Returns the number of moves done so far.
(count MovesThisTurn)	Returns the number of moves done so far in this turn.
(count Neighbours ...)	Returns the number of neighbours sites of a site.
(count Off ...)	Returns the number of off-diagonal sites of a site.
(count Orthogonal ...)	Returns the number of orthogonal sites of a site.
(count Pieces ...)	Returns the number of pieces of a player.
(count Pips)	Returns the total number of pips of all the dice.
(count Phases)	Returns the number of phases of the game.
(count Players)	Returns the number of players.
(count Rows)	Returns the number of rows of the board.
(count Sites in:...)	Returns the number of sites in a region function.
(count Stack ...)	Returns the number of pieces in stack(s) according to conditions.
(count Steps ...)	Returns the distance between two sites. The step between two sites can be defined with a specific relation type or with a Step moves ludeme.
(count StepsOnTrack ...)	Returns the number of steps between two sites in following a track.
(count Trials)	Returns the number of trials done so far (for match game).
(count Turns)	Returns the number of turns done so far.
(count Vertices)	Returns the number of vertices of the board.

5.2.5 Dice

The dice Int ludeme is composed in the current version of a single ludeme:

IntFunction	Description
(face)	Returns the current value of a dice according to the site state where it is placed.

5.2.6 Iterator

The iterator Int functions return an integer used as an iterator. These iterators can be used in ludemes looping through values such as (forEach ...) or for ludeme checking many sites in each direction such as many moves ludemes.

Here is the list of them:

IntFunction	Description
(between)	Returns the current value of the between iterator.
(edge)	Returns the current value of the edge iterator.
(from)	Returns the current value of the from iterator.
(hint)	Returns the current value of the hint iterator.
(level)	Returns the current value of the level iterator.
(pips)	Returns the current value of the pips iterator.
(player)	Returns the current value of the player iterator.
(site)	Returns the current value of the site iterator.
(team)	Returns the current value of the team iterator.
(to)	Returns the current value of the to iterator.
(track)	Returns the current value of the track iterator.
(value)	Returns the current value of the value iterator.

5.2.7 Last

The last Int ludeme returns integers information about the last state:

IntFunction	Description
(last From)	Returns the index of the 'from' site of the last decision move.
(last To)	Returns the index of the 'to' site of the last decision move.

When the optional parameter *afterConsequence* is true, the locations returned of the last move are after applying the consequences.

5.2.8 Match

The match Int ludeme is composed in the current version of a single ludeme and returns information about match:

IntFunction	Description
(matchScore ...)	Returns the score of a player in the match.

5.2.9   Size

The size Int ludeme is composed in the current version of a single ludeme and returns information about match:

IntFunction	Description
(size Array ...)	Returns the size of an IntArrayFunction.
(size Group ...)	Returns the size of the group of pieces starting from a specific site.
(size Large Piece ...)	Returns the number of sites occupied by a large piece.
(size Stack ...)	Returns the size of the stack placed on a site.
(size Territory ...)	Returns the number of sites enclosed by the pieces of a player.

5.2.10   Stacking

The stacking Int ludeme is composed in the current version of a single ludeme:

IntFunction	Description
(topLevel ...)	Returns the index of the topmost level of a stack placed at a specific site.

5.2.11   State

The state Int ludemes returns an integer value based on the current game state:

IntFunction	Description
(amount ...)	Returns the amount of a player set with the ludeme (set Amount ...).
(counter)	Returns the counter used in the state or set with (set Counter ...).
(mover)	Returns the index of the player who has to move.
(next)	Returns the index of the player who has to move in the next state.
(pot)	Returns the value of the pot set with the ludeme (set Pot ...).
(pre ...)	Returns the index of the player who has played the previous move.
(rotation ...)	Returns the index of the rotation of a piece placed on a site.
(score ...)	Returns the score of a player.
(state ...)	Returns the state of a piece placed on a site.
(var ...)	Returns the value of a variable var set in using the ludeme (set Var ...).
(what ...)	Returns the index of the piece placed on a site.

(who ...)	Returns the index of the player owning a piece placed on a site.

5.2.12   Tile

The tile Int ludeme is composed in the current version of a single ludeme:

IntFunction	Description
(pathExtent ...)	Returns the maximum extent of a path.

5.2.13   TrackSite

The trackSite Int ludemes return integer value related to a site belonging to a track:

IntFunction	Description
(trackSite Move ...)	Returns the index of the site from a site belonging to the track after a specific number of steps.
(trackSite EndTrack ...)	Returns the index of the site at the end of the track.
(trackSite FirstSite ...)	Returns the index of the first site verifying a condition.

5.2.14   Value

The value Int ludemes return integer value related to the keyword value:

IntFunction	Description
(value at:...)	Returns the value of a piece at a specific location.
(value Player)	Returns the value associated to a player set with (set Value Player ...).
(value Pending)	Returns the pending value of the state set with (set Pending ...).
(value Random ...)	Returns a random value within a range.

5.2.15   Constants

The Ludii's compiler can convert a few constants to integers:

Constants	Value
Off	-1
End	-2
Undefined	-1
Unused	-4
NoPiece	0
Infinity	1000000000

5.3 IntArrayFunction

The Int Array functions return an array of integer values. These functions are not commonly used because of the Region functions (see Section 5.4) which are dedicated to set of sites. However some ludemes refer directly to these functions. The next subsections give a short description of each of them.

5.3.1 Math

The mathematics Int Array functions contains classical mathematics operators returning an array of integers:

IntArrayFunction	Description
(difference ...)	Returns an array containing the integers in the first parameter that are not in the second parameter.
(intersection ...)	Returns an array containing the integers which are in each array in entry.
(results ...)	Returns an array of all the results of the function in parameter for each site 'from' to each site 'to'.
(union ...)	Returns an array containing the integers on each array in entry.

5.3.2 Players

The Players Int Array functions return an array of players indices.

IntArrayFunction	Description
(players Team....)	Returns the indices of all the players in a team.
(players ... of:...)	Returns the indices of players related to the player index specified.

5.3.3 Sizes

The Sizes Int Array functions return an array of sizes of regions.

IntArrayFunction	Description
(group ...)	Returns the sizes of the group computing according to the parameters.

5.3.4 State

The State Int Array functions return an array of integers depending on the current state.

IntArrayFunction	Description
(rotations ...)	Converts a list of facing directions to a list of integer directions according to the rotation of the piece placed on a site.

5.3.5 Values

The Values Int Array functions return an array of integers according to the parameter.

IntArrayFunction	Description
(values Remembered ...)	Returns the current list of values remembered in the state because of the use of the move (remember Value ...).

5.4 RegionFunction

The Region functions return a list of Integer corresponding to a list of indices of sites. An empty list is returned in case of no site in the result of a Region function.

5.4.1 Math

The mathematics Region functions are mathematics operators about set or about set of vertices/faces in a graph:

RegionFunction	Description
(difference ...)	Returns the sites in the first Region function parameter without the sites in the second Region function parameter.
(expand ...)	Returns the sites of a region expanding in one or all the directions with a specific distance.
(if ...)	Returns the result of a Region function among two according to a condition.
(intersection ...)	Returns the sites in common between many regions.
(union ...)	Returns all the sites of many regions.

5.4.2 Last

The last Region ludeme returns regions information about the last state:

RegionFunction	Description
(last Between)	Returns the indices of the 'between' sites of the last decision move.

5.4.3 ForEach

The forEach Region ludeme is used to filter a region or to create a region based on different player indices:

RegionFunction	Description
(forEach ...)	Returns the sites satisfying a constraint from a given region.
(forEach (players ...) ...)	Returns a region in iterating player indices.
(forEach Team)	Returns a region in iterating the teams.

With the ludeme (forEach ...), each site in the region is described by the ludeme (site). In the following example, the ludeme (forEach ...) returns the sites occupied by the next player that are not currently in a line of size 3 or more.

Ludeme 86
```
(forEach
    (sites Occupied by:Next)
    if:(not
        (is Line 3 through:(site))
    )
)
```

With the ludeme (forEach (players ...) ...), each player in the IntArrayFunction is described by the ludeme (player). In the following example, the ludeme (forEach (players ...) ...) returns the sites occupied by each player ally with the mover.

Ludeme 87
```
(forEach
    (players Ally of:(mover))
    (sites Occupied by:Player)
)
```

With the ludeme (forEach Team ...), each team is iterated with the ludeme (team). In the following example, the ludeme (forEach Team ...) returns the sites occupied by each player of each team.

Ludeme 88
```
(forEach Team
    (forEach (team)
        (sites Occupied by:Player)
    )
)
```

5.4.4 Sites

The (sites ...) Region ludeme can return many type of regions:

RegionFunction	Description
(sites)	Returns the sites iterated by ludemes iterated though many regions such as (forEach Group ...).
(sites ...)	Returns a set of sites corresponding to a list of indices or of a list of coordinates.
(sites Angled)	Returns all the angled edges of the board.
(sites Around ...)	Returns the sites around a given region or site according to specified directions and conditions.
(sites Axial)	Returns all the axial edges of the board.

(sites Between)	Returns all the 'between' sites of a set of moves.
(sites Board)	Returns all the sites of the board.
(sites Bottom)	Returns all the sites at the bottom of the board.
(sites Cell ...)	Returns all the vertices of a cell.
(sites Centre)	Returns all the sites in the centre of a board.
(sites Column ...)	Returns all the sites of a column.
(sites ConcaveCorners ...)	Returns all the concave corners sites of the board.
(sites ConvexCorners ...)	Returns all the convex corners sites of the board.
(sites Corners ...)	Returns all the corners sites of the board.
(sites Direction ...)	Returns the sites in a direction from a site.
(sites Distance ...)	Returns the sites at a specific distance from another.
(sites Edge ...)	Returns all the vertices of an edge.
(sites Empty)	Returns all the empty sites.
(sites From)	Returns all the 'from' sites of a set of moves.
(sites Hand ...)	Returns all the sites of a hand container.
(sites Hidden ...)	Returns all the sites invisible to a player
(sites Hidden Count ...)	Returns all the sites with the piece number hidden to a player
(sites Hidden Rotation ...)	Returns all the sites with the piece rotation hidden to a player
(sites Hidden Value ...)	Returns all the sites with the piece value hidden to a player
(sites Hidden State ...)	Returns all the sites with the state hidden to a player
(sites Hidden What ...)	Returns all the sites with the piece index hidden to a player
(sites Hidden Who ...)	Returns all the sites with the piece owner hidden to a player
(sites Hint ...)	Returns all the sites with a hint on them.
(sites Horizontal)	Returns all the horizontal edges of the board.
(sites Incident ...)	Returns the sites connected to a specific site.
(sites Inner)	Returns all the inner sites of the board.
(sites LargePiece ...)	Returns all the sites used by a large piece from the root.
(sites LastFrom)	Returns all the 'from' sites of the last move.
(sites LastTo)	Returns all the 'to' sites of the last move.
(sites Layer ...)	Returns all the sites of a layer.
(sites Left)	Returns all the sites at the leftmost of the board.
(sites LineOfPlay)	Returns all the sites in the line of play of domino games.
(sites LineOfSight ...)	Returns all the sites in a specific line of sight from a site.
(sites Loop ...)	Returns all the sites making a loop or inside the loop.
(sites Occupied ...)	Returns the sites occupied by pieces owned by a given player in a given container.

(sites Major)	Returns all the sites used as major generators to generate the graph.
(sites Minor)	Returns all the sites used as minor generators to generate the graph.
(sites Outer)	Returns all the outer sites of the board.
(sites P...)	Returns the sites corresponding to regions defined as owned by a given player in the equipment.
(sites Pattern ...)	Returns the sites in a specific pattern.
(sites Pending)	Returns all the sites in pending in the current state.
(sites Perimeter)	Returns all the perimeter sites of the board.
(sites Phase ...)	Returns all the sites in a specific phase.
(sites Playable)	Returns all the playable sites of a boardless game.
(sites Right)	Returns all the sites at the rightmost of the board.
(sites Random ...)	Returns a list of random sites.
(sites Row ...)	Returns all the sites of a row.
(sites Side ...)	Returns all the sites of a given side of the board.
(sites Slash)	Returns all the slash edges of the board.
(sites Slosh)	Returns all the slosh edges of the board.
(sites Start ...)	Returns the sites occupied by pieces of a given type in the initial state.
(sites State ...)	Returns all the sites with a specific state site.
(sites To)	Returns all the 'to' sites of a set of moves.
(sites ToClear)	Returns all the sites to remove the pieces on when the turn of the player will be over.
(sites Top)	Returns all the sites at the topmost of the board.
(sites Track ...)	Returns all the sites of a track.
(sites Vertical)	Returns all the vertical edges of the board.
(sites Walk ...)	Returns all the sites reached in following a walk.
(sites Winning ...)	Returns all the winning sites for a player.

5.5 DirectionFunctions

The Direction functions return a list of Absolute directions. They are used inside ludemes (generally Moves ludemes) to specify set of directions (e.g. (move Slide ...)).

DirectionFunctions	Description
(difference ...)	Returns the set of absolute directions of the first entry which are not in the second entry.
(directions ... from:... to:...)	Returns the direction between 'from' and 'to' if exists.

(directions ...)	Converts the corresponding set of absolute directions or relative directions in entry to absolute directions.
(directions Random ...)	Returns a random set of directions.
(if ...)	Returns a set of directions according to a condition.

5.6 RangeFunctions

The Range functions are used in some ludemes to specify a minimum and/or a maximum to obtain a range.

RangeFunction	Description
(exact ...)	To define a range of a single value.
(min ...)	To define a range with a minimum but no maximum.
(max ...)	To define a range with a maximum but no minimum.
(range ...)	To define a range with a minimum and a maximum.

6
Options

For many games there exist alternative rule sets and other variable aspects, such as different board sizes, number of pieces, starting positions, and so on. The Ludii game compiler supports an *option* mechanism to allow such alternatives for a game to be defined in a single description, to avoid the need to implement each one in its own file. Options are defined outside the main `game` ludeme but typically used within it. Options are typically declared directly below the `game ...)` ludeme, for clarity.

Options are instantiated at compile time and can be arbitrarily large, including choices between complete game descriptions if desired. Multiple options can be specified in combination, to give dozens or even hundreds of variant rule sets for a single game description.

Each set of options is declared with the `option` keyword and constitutes an option *category* with a number of option *items*. Each option item has a name and one or more *arguments* followed by a short description. The following example shows options for board sizes and for the ending rules:

> **❖ Ludeme 89**
> ```
> (option "Board Size" <Board> args:{ <size> } {
>     (item "5x5"    <5>   "The game is played on a 5x5 board.")
>     (item "10x10"  <10>  "The game is played on a 10x10 board.")
>     (item "19x19"  <19>  "The game is played on a 19x19 board.")*
> })
>
> (option "End Rules" <Result> args:{ <type>} {
>     (item "Standard" <Win>  "The player to reach the end wins.")*
>     (item "Misere"   <Loss> "The player to reach the end loses.")
> })
> ```

Option items are referenced in the game description by tag-argument pairs `<Tag:arg>`. However if the option has only one option, the parameter can be not specified. For example, this option call in a game description. The first one does not specify the parameter contrary

to the second one.

Ludeme 90

```
(board (hex Diamond <Board>))

(end
    (if
        (is Connected Mover)
        (result Mover <Result:type>)
    )
)
```

In order to specify the main/default option(s) of a game, an asterisk may optionally be appended to the end of each option item (e.g. option "19x19" or "Standard" in the examples). If no user-selected options are specified when a game is compiled, then the highest priority item within each category becomes the current option for that category.

The names of the option categories can be anything, however conventionally, we try to use the following ones when that's possible:

- "Board": To modify the kind of board, mainly for the GUI.

- "Board Size": To modify the size of the board.

- "Challenge": For the different challenge of a deduction puzzle.

- "End Rules": To modify the ending rules of the game.

- "Play Rules": To modify the playing rules of the game.

- "Players": To modify the number of players.

- "Start Rules": To modify the starting rules of the game.

- "Tiling": To modify the tiling of the board.

- "Turn Limit": To modify the turn limit.

- "Variant" or "Version": To modify many rule types in the same time.

6.1   Rulesets

In addition to the `option` mechanism, Ludii also supports a `ruleset` mechanism that allows user to declare custom rule sets defined by combinations of options.

In the following example, two rulesets are defined, one corresponding to a board of size 19x19 with the standard victory rules and another with a board of size 10x10 with the misere ending rules.

```
(rulesets {
        (ruleset "Ruleset/Standard" {
             "Board Size/19x19" "End Rules/Standard"
        })

        (ruleset "Ruleset/Misere" {
             "Board Size/10x10" "End Rules/Misere"
        })
})
```

The convention to name each *ruleset* is to start them by "Ruleset/" then the name of the ruleset.

<div align="right">

7
Defines

</div>

The *define* is a mechanism used to encapsulate a part of a description that can be repeated many times in the description, they can be viewed as a macro.

Defines are useful for:

- Simplifying game descriptions by wrapping complex ludeme structures into simple labels.

- Giving meaningful names to useful game concepts.

- Gathering repeated ludeme structures that are duplicated across multiple games into a single reusable definition.

When a game has to be compiled, Ludii first expands all the defines before compiling the game description. A define has a name and possibly can be parameterised. The parameters of a define are described with the symbol # followed by the indices of the parameters of the define: #1, #2, ..., #n for n parameters.

In the following example we define a hop move, which can be parameterised by the condition of the piece to jump, the effect to apply to it and the condition on the site to move to.

✿ Ludeme 92

```
(define "Hop"
    (move Hop
        (between
            if:#1
            #2
        )
        (to if:#3)
    )
)
```

To call a define, the name of the define is used at the beginning of the parenthesis, then each ludeme parameter value is provided such as in the following example of the hop move in the "Ball" piece jumping an enemy piece to capture it. However, if the middle parameter does

not have to be provided the symbol ~ can be used, such as in the example of the hop move in the "Disc" piece, jumping any enemy pieces without removing it.

> **✿ Ludeme 93**
>
> ```
> (piece "Ball"
>     ("Hop"
>         (is Enemy (who at:(between)))
>         (apply (remove (between)))
>         (is Empty (to))
>     )
> )
>
> (piece "Disc"
>     ("Hop"
>         (is Enemy (who at:(between)))
>         ~
>         (is Empty (to))
>     )
> )
> ```

As an advice for game designers, the define structure can also be used to help to make sense of some technical descriptions. For example, the state of a site or the pending values can be used for many purposes and may not be representative of what they really describe. However in using a define with a descriptive name, this is possible to give them a sense to make a better description. For example, in the description of Chess, the pending values are used to describe "En Passant" but that is clear only in using the following define

> **✿ Ludeme 94**
>
> ```
> (define "SetEnPassantLocation"
>     (then
>         (set Pending (ahead (last To) Backward))
>     )
> )
> ```

Defines can occur anywhere in the game description file even within ludemes but are typically at the top of the file, before the `game` ludeme, to impart some structures. Defines can be nested within other defines but not themselves if they have exactly the same parameters called. The convention is to name each *define* with a single compound word in "UpperCamelCase" format.

7.1 Ludemeplexes

Defines constitute *ludemeplexes*, which are useful structures of component *ludemes*[36]. These can be defined either:

- globally in their own .def file, in which case they become *known defines*, or
- locally within any .lud file in which they can be used.

---

[36]Ludemes are "game memes" or units of game-related information (Parlett, 2016).

The current list of known defines is available in the Ludii Language Reference (Browne et al., 2020).

8
Conclusion

The document will be periodically updated to reflect developments in the game logic of future Ludii releases, and in accordance with feedback from Ludii users. Please post any suggestions for improvements to the relevant Ludii Forum (Here) or email us at `ludii.games@gmail.com`.

9
Acknowledgements

European Research Council
Established by the European Commission

We would like to thank all the Ludii users who report errors, bugs or possible improvements of Ludii and its language. This work would not be possible without all their support and help.

References

Browne, C. (2008). *Automatic Generation and Evaluation of Recombination Games* (Ph.D. dissertation). QUT, Brisbane.

Browne, C. (2016). A Class Grammar for General Games. *Computers and Games*, *LNCS 10068*, 167–182.

Browne, C., Soemers, D. J. N. J., Piette, É., Stephenson, M., & Crist, W. (2020). *Ludii language reference.* https://ludii.games/downloads/LudiiLanguageReference.pdf.

Parlett, D. (2016). What's a Ludeme? *Game & Puzzle Design*, *2*(2), 81–84.

Piette, C., Piette, É., Stephenson, M., Soemers, D. J. N. J., & Browne, C. (2019). Ludii and XCSP: Playing and Solving Logic Puzzles. In *2019 IEEE Conference on Games (CoG)* (pp. 630–633).

Piette, É., Soemers, D. J. N. J., Stephenson, M., Sironi, C. F., Winands, M. H. M., & Browne, C. (2020). Ludii – The Ludemic General Game System. In G. D. Giacomo et al. (Eds.), *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)* (Vol. 325, p. 411-418). IOS Press.