Ludii Language Reference

Cameron Browne, Dennis J. N. J. Soemers,
Éric Piette, Matthew Stephenson and Walter Crist

Department of Data Science and Knowledge Engineering (DKE)

Maastricht University

Maastricht, the Netherlands

Ludii Version 1.0.7

September 18, 2020

Ludii Language Reference

This document provides full documentation for the game description language used by the Ludii general game system. Note that the majority of this document is automatically generated. More info on Ludii may be found on its website: `https://ludii.games/`. For questions or suggestions, please contact us on the Ludii forums (`https://ludii.games/forums/`), or send an email to ludii.games@gmail.com.

Contents

1 Introduction

This document provides a full reference for the game description language used to describe games for the Ludii general game system. Games in Ludii are described as *ludemes*, which may intuitively be understood to encapsulate simple concepts related to game rules or equipment. Every game description starts with a `game` ludeme, described in the form `(game ...)` in game description files. Here, the dots `(...)` are a placeholder for one or more arguments that are supplied to the `game` ludeme.

Arguments provided to ludemes may be:

- *Strings*: described in Section 1.1.
- *Booleans*: described in Section 1.2.
- *Integers*: described in Section 1.3.
- *Floats*: described in Section 1.4.
- *Other ludemes*: described throughout most of the other chapters of this document.

Part I describes all the ludemes that can be used in game descriptions. This is the most important part for writing new games that can be run in Ludii. Part II describes ludemes that can be used to add extra metadata to games. These are not strictly required for games to run, but can be used to provide additional information about games in Ludii, or to modify how they look in Ludii or how Ludii's AIs play them. More advanced language features are described in Part III.

1.1 Strings

Strings are simply snippets of text, typically used to assign names to pieces of game equipment, rules, or other concepts. Strings in game descriptions can be written by wrapping any snippet of text in a pair of double quotes. For instance, `"Pawn"` can be used to provide a name to a piece. By convention, the first symbol in a string is usually an uppercase character, but this is generally not required.

1.2 Booleans

There are two boolean values; `true` and `false`. They can be written as such in any game description file, without any additional notation.

1.3 Integers

Integers are numbers without a decimal component, such as `1`, `-1`, `100`, etc. They can simply be written as such, without any additional notation, in Ludii's game description language.

1.4 Floats

Floating point values are numbers with a decimal component, such as `0.5`, `-1.2`, `5.5`, etc. If a ludeme expects a floating point value as an argument, it must always be written to include a dot. For example, `1` cannot be interpreted as a floating point value, but `1.0` can.

Part I

Ludemes

2
Game

The `game` ludeme defines all aspects of the current game being played, including its equipment and rules. This ludeme is the root of the *ludemeplex* (i.e. structured tree of ludemes) that makes up this game.

2.1 Game

The base `game` ludeme describes an instance of a single game.

2.1.1 game

Defines the main ludeme that describes the players, mode, equipment and rules of a game.

Format

```
(game <string> [<players>] [<mode>] [<equipment>] [<rules>])
```

where:

- `<string>`: The name of the game.
- `[<players>]`: The players of the game [(players 2)].
- `[<mode>]`: The mode of the game [Alternating].
- `[<equipment>]`: The equipment of the game [One square board of size 3 and one ball per player].
- `[<rules>]`: The rules of the game [The rules of Tic-Tac-Toe].

Example

```
(game "Tic-Tac-Toe"
    (players 2)
    (equipment
        {
            (board (square 3))
            (piece "Disc" P1)
            (piece "Cross" P2)
        }
    )
    (rules
        (play (move Add (to (sites Empty))))
        (end (if (is Line 3) (result Mover Win)))
    )
)
```

Remarks

If no rules and no equipment are defined, the default game is Tic-Tac-Toe.

2.2 Functions - Dim - Math

Math functions return an integer value based on given inputs.

2.2.1 abs

Return the absolute value of a dim.

Format

```
(abs <dimFunction>)
```

where:

- `<dimFunction>`: The value.

Example

```
(abs (- 8 5))
```

2.2.2 + (add)

Adds many values.

Format

To add two values.

```
(+ <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The first value.
- `<dimFunction>`: The second value.

To add all the values of a list.

```
(+ {<dimFunction>})
```

where:

- `{<dimFunction>}`: The list of the values.

Examples

```
(+ 5 2)

(+ {10 2 5})
```

Remarks

This is used to add many values.

2.2.3 / (div)

To divide a value by another.

Format

```
(/ <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The value to divide.
- `<dimFunction>`: To divide by b.

Example

```
(/ 4 2)
```

Remarks

The result will be an integer.

2.2.4 max

Returns the maximum of two specified values.

Format

```
(max <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The first value.
- `<dimFunction>`: The second value.

Example

```
(max 9 3)
```

2.2.5 min

Returns the minimum of two specified values.

Format

```
(min <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The first value.
- `<dimFunction>`: The second value.

Example

```
(min 20 3)
```

2.2.6 ∗ (mul)

Returns to multiple of values.

Format

For a product of two values.

```
(∗ <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The first value.
- `<dimFunction>`: The second value.

For a product of many values.

```
(∗ {<dimFunction>})
```

where:

- `{<dimFunction>}`: The list of values.

Examples

```
(* 6 2)

(* {3 2 5})
```

2.2.7 ∧ (pow)

Computes the first parameter to the power of the second parameter.

Format

```
(∧ <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The value.
- `<dimFunction>`: The power.

Example

```
(^ 2 2)
```

2.2.8 − (sub)

Returns the subtraction A minus B.

Format

```
(− <dimFunction> <dimFunction>)
```

where:

- `<dimFunction>`: The value A.
- `<dimFunction>`: The value B.

Example

```
(- 5 1)
```

2.3 Match

Matches are composed of multiple *instances* of component games. Each match maintains additional state information beyond that stored for each of its component games, and is effectively a super-game whose result is determined by the results of its sub-games.

2.3.1 games

Defines the games used in a match.

Format

```
(games (<subgame> | {<subgame>}))
```

where:

- `<subgame>`: The game that makes up the subgames of the match.
- `{<subgame>}`: The games that make up the subgames of the match.

Example

```
(games
    {
        (subgame "Tic-Tac-Toe" next:1)
        (subgame "Yavalath" next:2)
        (subgame "Breakthrough" next:0)
    }
)
```

2.3.2 match

Defines a match made up of a series of subgames.

Format

```
(match <string> [<players>] <games> <end>)
```

where:

- `<string>`: The name of the match.
- `[<players>]`: The players of the match [(players 2)].
- `<games>`: The different subgames that make up the match.
- `<end>`: The end rules of the match.

Example

```
(match "Match"
    (players 2)
    (games
        {
            (subgame "Tic-Tac-Toe" next:1)
            (subgame "Yavalath" next:2)
            (subgame "Breakthrough" next:0)
        }
    )
    (end
        {
            (if
                (and (= (count Trials) 3) (> (matchScore P1) (matchScore P2)))
                (result P1 Win)
            )
            (if
                (and (= (count Trials) 3) (< (matchScore P1) (matchScore P2)))
                (result P2 Win)
            )
            (if
                (and (= (count Trials) 3) (= (matchScore P1) (matchScore P2)))
                (result P1 Draw)
            )
        }
    )
)
```

2.3.3 subgame

Defines an instance game of a match.

Format

```
(subgame <string> [<string>] [next:<int>] [result:<int>])
```

where:

- `<string>`: The name of the game instance.
- `[<string>]`: The option of the game instance.
- `[next:<int>]`: The index of the next instance.
- `[result:<int>]`: The score result for the match when game instance is over.

Example

```
(subgame "Tic-Tac-Toe")
```

2.4 Mode

The *mode* of a game refers to the way it is played. Ludii supports the following modes of play:

- *Alternating*: Players take turns making discrete moves.
- *Simultaneous*: Players move at the same time.

2.4.1 mode

Describes the mode of play.

Format

```
(mode [<modeType>])
```

where:

- `[<modeType>]`: The mode of the game.

Example

```
(mode Simultaneous)
```

2.5 Players

The *players* of a game are the entities that compete within the game according to its rules. Players can be:

- *Human*: i.e. you!
- *AI*: Artificial intelligence agents.
- *Remote*: Remote players over a network, which may be Human or AI.

Each player has a name and a number according to the default play order. The Neutral player (index 0) is used to denote equipment that belongs to no player, and to make moves associated with the environment rather than any actual player. The Shared player (index $N+1$ where N is the number of players) is used to denote equipment that belongs to all players. The actual players are denoted P1, P2, P3, ... in game descriptions.

2.5.1 player

A player of the game.

Format

```
(player [<string>] [<directionFacing>] [<moves>])
```

where:

- [<string>]: The name of the player.
- [<directionFacing>]: The direction of the pieces of the player.
- [<moves>]: The moves associated with all the component owned by that player.

Example

```
(player N)
```

Remarks

To define a player with a specific name or direction.

2.5.2 players

Defines the players of the game.

Format

To define a set of many players with specific data for each.

```
(players {<player>})
```

where:

- `{<player>}`: The list of players.

To define a set of many players with the same data for each.

```
(players int [<moves>])
```

where:

- `int`: The number of players.
- `[<moves>]`: The moves associated with all the component owned by that player.

Examples

```
(players { (player N) (player S) })

(players 2)
```

3
Equipment

The *equipment* of a game refers to the items with which it is played. These include *components* such as pieces, cards, tiles, dice, etc., and *containers* that hold the components, such as boards, decks, player hands, etc. Each container has an underlying graph that defines its playable sites and adjacencies between them.

3.1 Equipment

The following ludemes describe the equipment used to play the game.

3.1.1 equipment

Defines the equipment list of the game.

Format

```
(equipment {<item>})
```

where:

- {<item>}: The items (container, component etc.).

Example

```
(equipment
    {
        (board (square 3))
        (piece "Disc" P1)
        (piece "Cross" P2)
    }
)
```

Remarks

To define the items (container, component etc.) of the game. Any type of component or container described in this chapter may be used as an `<item>` type.

3.2 Component

Components correspond to physical pieces of equipment used in games, other than boards. For example: pieces, dice, etc. All types of components listed in this section may be used for `<item>` parameters in Equipment definitions.

3.2.1 card

Defines a card with specific properties such as the suit or the rank of the card in the deck.

Format

```
(card <string> <roleType> <cardType> rank:int value:int trumpRank:int
     trumpValue:int suit:int [<moves>])
```

where:

- `<string>`: The name of the card.
- `<roleType>`: The owner of the card.
- `<cardType>`: The type of a card chosen from the possibilities in the CardType ludeme.
- `rank:int`: The rank of the card in the deck.
- `value:int`: The value of the card.
- `trumpRank:int`: The trump rank of the card in the deck.
- `trumpValue:int`: The trump value of the card.
- `suit:int`: The suit of the card.
- `[<moves>]`: The moves associated with the component.

Example

```
(card "Card" Shared King rank:6 value:4 trumpRank:3 trumpValue:4 suit:1)
```

Remarks

This ludeme creates a specific card. If this ludeme is used with no deck defined, the generated card will be not included in a deck by default. See also Deck ludeme.

3.2.2 component

Defines a component.

3.2.3 die

Defines a single non-stochastic die used as a piece.

Format

```
(die <string> <roleType> numFaces:int [<directionFacing>] [int]
     [<moves>])
```

where:

- `<string>`: The name of the die.
- `<roleType>`: The owner of the die.
- `numFaces:int`: The number of faces of the die.
- `[<directionFacing>]`: The direction of the component.
- `[int]`: The value of the component.
- `[<moves>]`: The moves associated with the component.

Example

```
(die "Die6" All numFaces:6)
```

Remarks

The die defined with this ludeme will be not included in a dice container and cannot be rolled with the roll ludeme, but can be turned to show each of its faces.

3.2.4 piece

Defines a piece.

Format

```
(piece <string> [<roleType>] [<directionFacing>] [value:int] [<flips>]
     [<moves>])
```

where:

- `<string>`: The name of the piece.
- `[<roleType>]`: The owner of the piece [Each].
- `[<directionFacing>]`: The direction of the piece.
- `[value:int]`: The value of the piece.
- `[<flips>]`: The corresponding values to flip, e.g. (flip 1 2) 1 is flipped to 2 and 2 is flipped to 1.

- [<moves>]: The moves associated with the piece.

Examples

```
(piece "Pawn" Each)

(piece "Disc" Neutral (flips 1 2))

(piece "Dog" P1 (step (to if:(is Empty (to)))))
```

Remarks

Useful to create a pawn, a disc or a representation of an animal for example.

3.3 Component - Tile

Tiles are (typically flat) pieces that completely fill the cells they are placed in, and may have additional decorations (such as paths) drawn on them.

3.3.1 domino

Defines a single domino.

Format

```
(domino <string> <roleType> value:int value2:int [<moves>])
```

where:

- `<string>`: The name of the domino.
- `<roleType>`: The owner of the domino.
- `value:int`: The first value of the domino.
- `value2:int`: The second value of the domino.
- `[<moves>]`: The moves associated with the component.

Example

```
(domino "Domino45" Shared value:4 value2:5)
```

Remarks

The domino defined with this ludeme will be not included in the dominoes container by default and so cannot be shuffled with other dominoes.

3.3.2 path

Defines the internal path of a tile component.

Format

```
(path from:int [slotsFrom:int] to:int [slotsTo:int] colour:int)
```

where:

- `from:int`: The "from" side of the connection.
- `[slotsFrom:int]`: The slot of the "from" side [0].
- `to:int`: The "to" side of the connection.

- [slotsTo:int]: The slot of the "to" side [0].
- colour:int: The colour of the connection.

Example

```
(path from:0 to:2 colour:1)
```

Remarks

To define the path of the internal connection of a tile component. The number side 0 = the first direction of the tiling, in general 0 = North.

3.3.3 tile

Defines a tile, a component following the tiling with internal connection.

Format

```
(tile <string> [<roleType>] ([{<stepType>}] | [{{<stepType>}}])
     [numSides:int] ([slots:{int}] | [slotsPerSide:int]) [{<path>}]
     [<flips>] [<moves>])
```

where:

- <string>: The name of the tile.
- [<roleType>]: The owner of the tile.
- [{<stepType>}]: A turtle graphics walk to define the shape of a large tile.
- [{{<stepType>}}]: Many turtle graphics walks to define the shape of a large tile.
- [numSides:int]: The number of sides of the tile.
- [slots:{int}]: The number of slots for each side.
- [slotsPerSide:int]: The number of slots for each side if this is the same number for each side [1].
- [{<path>}]: The connection in the tile.
- [<flips>]: The corresponding values to flip, e.g. (flip 1 2) 1 is flipped to 2 and 2 is flipped to 1.
- [<moves>]: The associated moves of this component.

Example

```
(tile
    "TileX"
    numSides:4
    { (path from:0 to:2 colour:1) (path from:1 to:3 colour:2) }
)
```

3.4 Container - Board

This section lists a variety of basic board types. All of these types may be used for `<item>` parameters in Equipment definitions.

3.4.1 board

Defines a board by its graph, consisting of vertex locations and edge pairs.

Format

```
(board <graphFunction> ([<track>] | [{<track>}]) [use:<siteType>])
```

where:

- `<graphFunction>`: The graph function used to build the board.
- `[<track>]`: The track on the board.
- `[{<track>}]`: The tracks on the board.
- `[use:<siteType>]`: Graph element type to use by default [Cell].

Example

```
(board
    (graph
        vertices:{
            {1 0} {2 0} {0 1} {1 1} {2 1} {3 1} {0 2} {1 2} {2 2} {3 2} {1 3}
            {2 3}
        }
        edges:{
            {0 2} {0 3} {3 2} {3 4} {1 4} {4 5} {1 5} {3 7} {4 8} {6 7} {7 8}
            {8 9} {6 10} {11 9} {10 7} {11 8}
        }
    )
)
```

Remarks

Useful to define any board not following a specific tiling/shape. The custom board uses all the intercardinal directions. The cells of the board are currently not computed, but will be in a future version. By default a board is played on the vertices.

3.4.2 boardless

Defines a boardless container growing in function of the pieces played.

Format

```
(boardless <tilingBoardlessType>)
```

where:

- `<tilingBoardlessType>`: The tiling of the boardless container.

Example

```
(boardless Hexagonal)
```

Remarks

The playable sites of the board will be all the sites adjacent to the places already played/placed. No pregeneration is computed on the graph except the centre.

3.4.3 track

Defines a named track for a container, which is typically the board.

Format

```
(track <string> ({int} | <string>) [loop:<boolean>] ([int] |
     [<roleType>]) [directed:<boolean>])
```

where:

- `<string>`: The name of the track.
- `{int}`: List of integers describing board site indices.
- `<string>`: Description including site indices and cardinale directions (N, E, S, W).
- `[loop:<boolean>]`: True if the track is a loop [false].
- `[int]`: The owner of the track [0].
- `[<roleType>]`: The role of the owner of the track [Neutral].
- `[directed:<boolean>]`: True if the track is directed [false].

Examples

```
(track "Track" "1,E,N,W" loop:true)

(track "Track1" {6 12..7 5..0 13..18 20..25 End } P1 directed:true)

(track "Track1" "20,3,W,N1,E,End" P1 directed:true)
```

Remarks

Tracks are typically used for race games, or any game in which pieces move around a track. A number after a direction indicates the number of steps in that direction. For example, "N1,E3" means that track goes North for one step then turns East for three steps.

3.5 Container - Board - Custom

This section lists a variety of customised, special board types. All of these types may be used for `<item>` parameters in Equipment definitions.

3.5.1 surakartaBoard

Defines a Surakarta-style board.

Format

```
(surakartaBoard <graphFunction> [loops:int] [from:int])
```

where:

- `<graphFunction>`: The graph function used to build the board.
- `[loops:int]`: Number of loops, i.e. special capture tracks [(minDim - 1) / 2].
- `[from:int]`: Which row to start loops from [1].

Example

```
(surakartaBoard (square 6) loops:2)
```

Remarks

Surakata-style boards have loops that pieces must travel around in order to capture other pieces. The following board shapes are supported: Square, Rectangle, Hexagon, Triangle.

3.6 Container - Board - Puzzle

This section lists a variety of specialised Board types for puzzles. All of these types may be used for `<item>` parameters in Equipment definitions.

3.6.1 puzzleBoard

Generates a puzzle board.

Format

```
(puzzleBoard <graphFunction> (<values> | {<values>}) [use:<siteType>])
```

where:

- `<graphFunction>`: The graph function used to build the board.
- `<values>`: The range values of a graph element.
- {`<values>`}: The range values of many graph elements.
- `[use:<siteType>]`: Graph element type to use by default [Cell].

Example

```
(puzzleBoard (square 9) (values Cell (range 1 9)))
```

Remarks

Puzzle boards are used for deduction puzzles. The state model for these puzzles is a Constraint Satisfaction Problem (CSP) model, possibly with a variable for each graph elements (i.e. vertex, edge and face), each with a range of possible values.

3.7 Container - Other

This section contains various types of containers (which can hold components) other than board types; these are often pieces, cards, etc. that are held by players in their hands. All of these types may be used for `<item>` parameters in Equipment definitions.

3.7.1 deck

Generates a deck of cards.

Format

```
(deck [<roleType>] [cardsBySuit:int] [suits:int] [{<card>}])
```

where:

- `[<roleType>]`: The owner of the deck [Shared].
- `[cardsBySuit:int]`: The number of cards per suit [13].
- `[suits:int]`: The number of suits in the deck [4].
- `[{<card>}]`: Specific data about each kind of card for each suit.

Examples

```
(deck)

(deck
    {
        (card Seven rank:0 value:0 trumpRank:0 trumpValue:0)
        (card Eight rank:1 value:0 trumpRank:1 trumpValue:0)
        (card Nine rank:2 value:0 trumpRank:6 trumpValue:14)
        (card Ten rank:3 value:10 trumpRank:4 trumpValue:10)
        (card Jack rank:4 value:2 trumpRank:7 trumpValue:20)
        (card Queen rank:5 value:3 trumpRank:2 trumpValue:3)
        (card King rank:6 value:4 trumpRank:3 trumpValue:4)
        (card Ace rank:7 value:11 trumpRank:5 trumpValue:11)
    }
)
```

3.7.2 dice

Generates a set of dice.

Format

```
(dice [d:int] ([faces:{int}] | [from:int]) [<roleType>] num:int
     [biased:{int}])
```

where:

- `[d:int]`: The number of faces of the die [6].
- `[faces:{int}]`: The values of each face.
- `[from:int]`: The starting value of each die [1].
- `[<roleType>]`: The owner of the dice [Shared].
- `num:int`: The number of dice in the set.
- `[biased:{int}]`: The biased values of each die.

Example

```
(dice d:2 from:0 num:4)
```

Remarks

Used for any dice game to define a set of dice. Only the set of dice can be rolled.

3.7.3 hand

Defines a hand of a player.

Format

```
(hand <roleType> [size:int])
```

where:

- `<roleType>`: The owner of the hand.
- `[size:int]`: The numbers of sites in the hand.

Example

```
(hand Each size:5)
```

Remarks

For any game with components outside of the board.

3.8 Other

This section describes other types of `Equipment` that the user can declare, apart from containers and components. These include Dominoes sets, `Hints` for puzzles, integer `Maps` and `Regions`, as follows.

3.8.1 dominoes

Defines a dominoes set.

Format

```
(dominoes [upTo:int])
```

where:

- `[upTo:int]`: The number of dominoes [6].

Example

```
(dominoes)
```

3.8.2 hints

Defines the hints of a deduction puzzle.

Format

```
(hints [<string>] {<hint>} [<siteType>])
```

where:

- `[<string>]`: The name of these hints.
- `{<hint>}`: The different hints.
- `[<siteType>]`: The graph element type of the sites.

Example

```
(hints
    {
        (hint {0 5 10 15} 3)
        (hint {1 2 3 4} 4)
        (hint {6 11 16} 3)
        (hint {7 8 9 12 13 14} 4)
        (hint {17 18 19} 3)
        (hint {20 21 22} 3)
        (hint {23 24} 1)
    }
)
```

Remarks

Used for any deduction puzzle with hints.

3.8.3 map

Defines a map between two locations or integers.

Format

For map of pairs.

(map [<string>] {<pair>})

where:

- [<string>]: The name of the map ["Map"].
- {<pair>}: The pairs of each map.

For map between integers.

(map [<string>] {int} {int})

where:

- [<string>]: The name of the map ["Map"].
- {int}: The keys of the map.
- {int}: The values of the map.

Examples

```
(map "Entry" { (pair P1 "D1") (pair P2 "E8") (pair P3 "H4") (pair P4 "A5") })

(map { (pair 5 19) (pair 7 9) (pair 34 48) (pair 36 38) })

(map
    {
        (pair P1 P4)
        (pair P2 P5)
        (pair P3 P6)
        (pair P4 P1)
        (pair P5 P2)
        (pair P6 P3)
    }
)

(map {1..9} { 1 2 4 8 16 32 64 128 256 })
```

Remarks

Used to map a site to another or to map an integer to another.

3.8.4 regions

Defines a static region on the board.

Format

```
(regions [<string>] [<roleType>] ({int} | <region> | {<region>} |
     <regionTypeStatic> | {<regionTypeStatic>}) [<string>])
```

where:

- [<string>]: The name of the region ["Region" + owner index].
- [<roleType>]: The owner of the region [P1].
- {int}: The sites included in the region.
- <region>: The region function corresponding to the region.
- {<region>}: The region functions corresponding to the region.
- <regionTypeStatic>: Pre-computed static region corresponding to this region.
- {<regionTypeStatic>}: Pre-computed static regions corresponding to this region.
- [<string>]: Name of this hint region (for deduction puzzles).

Examples

```
(regions P1 { (sites Side NE) (sites Side SW) })

(regions "Replay" {14 24 43 53})

(regions "Traps" (sites {"C3" "C6" "F3" "F6"}))
```

Remarks

Used when regions can be owned by players.

4
Graph Functions

Graph functions are ludemes that define operations that can be applied to arbitrary graph objects. These are typically used to transform or modify simpler graphs into more complex ones, for use as game baords.

4.1 Generators - Basis - Brick

This section contains the boards based on a square tiling.

4.1.1 brick

Defines a board on a brick tiling using 1x2 rectangular brick tiles.

Format

```
(brick [<brickShapeType>] <dimFunction> [<dimFunction>]
     [trim:<boolean>])
```

where:

- `[<brickShapeType>]`: Board shape [Square].
- `<dimFunction>`: First board dimension (size or number of rows).
- `[<dimFunction>]`: Second dimension (columns) [rows].
- `[trim:<boolean>]`: Whether to clip exposed half bricks [false].

Example

```
(brick Diamond 4 trim:true)
```

4.1.2 brickShapeType

Defines known shapes for the square tiling.

Value	Description
Square	Square board shape.
Rectangle	Rectangular board shape.
Diamond	Diamond board shape.
Prism	Prism board shape.
Spiral	Spiral board shape.
Limping	Alternating sides are staggered.

4.2 Generators - Basis - Celtic

This section contains boards based on a Celtic knotwork designs.

4.2.1 celtic

Defines a board based on Celtic knotwork.

Format

For defining a celtic tiling with the number of rows and the number of columns.

```
(celtic <dimFunction> [<dimFunction>])
```

where:

- `<dimFunction>`: Number of rows.
- `[<dimFunction>]`: Number of columns.

For defining a celtic tiling with a polygon or the number of sides.

```
(celtic (<poly> | {<dimFunction>}))
```

where:

- `<poly>`: Points defining the board shape.
- `{<dimFunction>}`: Length of consecutive sides of outline shape.

Examples

```
(celtic 3)

(celtic (poly { { 1 2} { 1 6 } { 3 6 } { 3 4 } { 4 4 } { 4 2 } }))

(celtic {4 3 -1 2 3})
```

Remarks

Celtic knotwork typically has a small number of continuous paths crossing the entire area – usually just one – making these designs an interesting choice for path-based games.

4.3 Generators - Basis - Hex

This section contains the boards based on a hexagonal tiling.

4.3.1 hex

Defines a board on a hexagonal tiling.

Format

For defining a hex tiling with two dimensions.

```
(hex [<hexShapeType>] <dimFunction> [<dimFunction>])
```

where:

- `[<hexShapeType>]`: Board shape [Hexagon].
- `<dimFunction>`: Primary board dimension; cells or vertices per side.
- `[<dimFunction>]`: Secondary Board dimension; cells or vertices per side.

For defining a hex tiling with a polygon or the number of sides.

```
(hex (<poly> | {<dimFunction>}))
```

where:

- `<poly>`: Points defining the board shape.
- `{<dimFunction>}`: Side lengths around board in clockwise order.

Examples

```
(hex 5)

(hex Diamond 11)

(hex Rectangle 4 6)

(hex (poly { { 1 2} { 1 6 } { 3 6 } }))

(hex {4 3 -1 2 3})
```

4.3.2 hexShapeType

Defines known shapes for the hexagonal tiling.

Value	Description

NoShape	No shape; custom graph.
Square	Square board shape.
Rectangle	Rectangular board shape.
Diamond	Diamond board shape.
Triangle	Triangular board shape.
Hexagon	Hexagonal board shape.
Star	Multi-pointed star shape.
Limping	Alternating sides are staggered.
Prism	Diamond shape extended vertically.

4.4 Generators - Basis - Morris

This section contains the boards based on the morris tiling. These are boards with concentric square rings joined by edges and with an empty middle, used for Morris games.

4.4.1 morris

Defines a Morris-style board.

Format

```
(morris <dimFunction> [joinCorners:<boolean>])
```

where:

- `<dimFunction>`: Number of concentric square rings.
- `[joinCorners:<boolean>]`: Whether to join corners with diagonal connections [false].

Examples

```
(morris 2)

(morris 3 joinCorners:true)
```

Remarks

Morris boards have concentric square rings joined by edges and an empty middle. Morris games are typically played on the vertices not the cells.

4.5 Generators - Basis - Quadhex

This section contains the boards based on the "quadhex" tiling. This is a hexagon tessellated by quadrilaterals, as used for the Three Player Chess board.

4.5.1 quadhex

Defines a "quadhex" board.

Format

```
(quadhex <dimFunction> [thirds:<boolean>])
```

where:

- `<dimFunction>`: Number of layers.
- `[thirds:<boolean>]`: Whether to split the board into three-subsections [false].

Example

```
(quadhex 4)
```

Remarks

The quadhex board is a hexagon tessellated by quadrilaterals, as used for the Three Player Chess board. The number of cells per side will be twice the number of layers.

4.6 Generators - Basis - Square

This section contains the boards based on a square tiling.

4.6.1 diagonalsType

Defines how to handle diagonal relations on the Square tiling.

Value	Description
Implied	Diagonal connections (not edges) between opposite corners.
Solid	Solid edges between opposite diagonals, which split the square into four triangles.
Alternating	Every second diagonal is a solid edge, as per Alquerque boards.
Concentric	Concentric diagonal rings from the centre.
Radiating	Diagonals radiating from the centre.

4.6.2 square

Defines a board on a square tiling.

Format

For defining a square tiling with the dimension.

```
(square [<squareShapeType>] <dimFunction> ([diagonals:<diagonalsType>]
    | [pyramidal:<boolean>]))
```

where:

- `[<squareShapeType>]`: Board shape [Square].
- `<dimFunction>`: Board dimension; cells or vertices per side.
- `[diagonals:<diagonalsType>]`: How to handle diagonals between opposite corners [Implied].
- `[pyramidal:<boolean>]`: Whether this board allows a square pyramidal stacking.

For defining a square tiling with a polygon or the number of sides.

```
(square (<poly> | {<dimFunction>}) [diagonals:<diagonalsType>])
```

where:

- `<poly>`: Points defining the board shape.
- `{<dimFunction>}`: Length of consecutive sides of outline shape.
- `[diagonals:<diagonalsType>]`: How to handle diagonals between opposite corners [Implied].

Examples

```
(square Diamond 4)

(square (poly { { 1 2} { 1 6 } { 3 6 } { 3 4 } { 4 4 } { 4 2 } }))

(square {4 3 -1 2 3})
```

4.6.3 squareShapeType

Defines known shapes for the square tiling.

Value	Description
NoShape	No shape; custom graph.

Square	Square board shape.
Rectangle	Rectangular board shape.
Diamond	Diamond board shape.
Limping	Alternating sides are staggered.

4.7 Generators - Basis - Tiling

This section defines the supported geometric board tilings, apart from regular tilings.

4.7.1 tiling

Defines a board graph by a known tiling and size.

Format

For defining a tiling with two dimensions.

```
(tiling <tilingType> <dimFunction> [<dimFunction>])
```

where:

- `<tilingType>`: Tiling type.
- `<dimFunction>`: Number of sites along primary board dimension.
- `[<dimFunction>]`: Number of sites along secondary board dimension [same as primary].

For defining a tiling with a polygon or the number of sides.

```
(tiling <tilingType> (<poly> | {<dimFunction>}))
```

where:

- `<tilingType>`: Tiling type.
- `<poly>`: Points defining the board shape.
- {`<dimFunction>`}: Side lengths around board in clockwise order.

Examples

```
(tiling T3636 3)

(tiling T3636 (poly { { 1 2} { 1 6 } { 3 6 } }))

(tiling T3636 {4 3 -1 2 3})
```

4.7.2 tilingType

Defines known tiling types for boards (apart from regular tilings).

Value	Description

T31212	Semi-regular tiling made up of triangles and dodecagons.
T3464	Rhombitrihexahedral tiling (e.g. Kensington).
T488	Semi-regular tiling made up of octagons with squares in the interstitial gaps.
T33434	Semi-regular tiling made up of squares and pairs of triangles.
T33336	Semi-regular tiling made up of triangles arond hexagons.
T33344	Semi-regular tiling made up of alternating rows of squares and triangles.
T3636	Semi-regular tiling made up of triangles and hexagons.
T4612	Semi-regular tiling made up of squares, hexagons and dodecagons.
T333333_33434	Tiling 3.3.3.3.3.3,3.3.4.3.4.

4.8 Generators - Basis - Tiling - Tiling3464

This section contains the boards based on a rhombitrihexahedral tiling (semi-regular tiling 3.4.6.4), such as the tiling used for the Kensington board.

4.8.1 tiling3464ShapeType

Defines known shapes for the rhombotrihexahedral (semi-regular 3.4.6.4) tiling.

Value	**Description**
Custom	Custom board shape.
Square	Square board shape.
Rectangle	Rectangular board shape.
Diamond	Diamond board shape.
Prism	Diamond board shape extended vertically.
Triangle	Triangular board shape.
Hexagon	Hexagonal board shape.
Star	Multi-pointed star shape.
Limping	Alternating sides are staggered.

4.9 Generators - Basis - Tri

This section contains the boards based on a triangular tiling.

4.9.1 tri

Defines a board on a triangular tiling.

Format

For defining a tri tiling with two dimensions.

```
(tri [<triShapeType>] <dimFunction> [<dimFunction>])
```

where:

- `[<triShapeType>]`: Board shape [Triangle].
- `<dimFunction>`: Board dimension; cells or vertices per side.
- `[<dimFunction>]`: Board dimension; cells or vertices per side.

For defining a tri tiling with a polygon or the number of sides.

```
(tri (<poly> | {<dimFunction>}))
```

where:

- `<poly>`: Points defining the board shape.
- `{<dimFunction>}`: Side lengths around board in clockwise order.

Examples

```
(tri 8)

(tri Hexagon 3)

(tri (poly { { 1 2} { 1 6 } { 3 6 } { 3 4 } }))

(tri {4 3 -1 2 3})
```

4.9.2 triShapeType

Defines known shapes for the triangular tiling.

Value	Description
NoShape	No shape; custom graph.

Square	Square board shape.
Rectangle	Rectangular board shape.
Diamond	Diamond board shape.
Triangle	Triangular board shape.
Hexagon	Hexagonal board shape.
Star	Multi-pointed star shape.
Limping	Alternating sides are staggered.
Prism	Diamond shape extended vertically.

4.10 Generators - Shape

This section contains different types of board shapes.

4.10.1 circle

Defines a board based on a circular tiling.

Format

```
(circle {<dimFunction>} [stagger:<boolean>])
```

where:

- `{<dimFunction>}`: Number of cells per concentric ring.
- `[stagger:<boolean>]`: Whether to stagger cells in adjacent rings [false].

Examples

```
(circle {8})

(circle {0 8})

(circle {1 4 8} stagger:true)
```

Remarks

Circular tilings are centred around a "pivot" point. The first ring value should be 0 (centre point) or 1 (centre cell).

4.10.2 rectangle

Defines a rectangular board.

Format

```
(rectangle <dimFunction> [<dimFunction>] [diagonals:<diagonalsType>])
```

where:

- `<dimFunction>`: Number of rows.
- `[<dimFunction>]`: Number of columns.
- `[diagonals:<diagonalsType>]`: Which type of diagonals to create, if any.

Example

```
(rectangle 4 6)
```

4.10.3 repeat

Repeats specified shape(s) to define the board tiling.

Format

```
(repeat <dimFunction> <dimFunction> step:{{<float>}} (<poly> |
     {<poly>}))
```

where:

- `<dimFunction>`: Number of rows to repeat.
- `<dimFunction>`: Number of columns to repeat.
- `step:{{<float>}}`: Vectors defining steps to the next column and row.
- `<poly>`: The shape to repeat.
- `{<poly>}`: The set of shapes to repeat.

Example

```
(repeat
    3
    4
    step:{ { -.5 .75} { 1 0 } }
    (poly { { 0 0} { 0 1 } { 1 1 } { 1 0 } })
)
```

4.10.4 shape

Defines simple uniform polygon shapes.

Format

```
(shape [Star] <dimFunction>)
```

where:

- `<dimFunction>`: Number of sides.

Example

```
(shape 3)
```

4.10.5 shapeStarType

Defines star shape types for known board types.

Value	Description
Star	Multi-pointed star shape.

4.10.6 spiral

Defines a board based on a spiral tiling, e.g. the Mehen board.

Format

```
(spiral turns:<dimFunction> sites:<dimFunction> [clockwise:<boolean>])
```

where:

- `turns:<dimFunction>`: Number of turns of the spiral.
- `sites:<dimFunction>`: Number of sites to generate in total.
- `[clockwise:<boolean>]`: Whether the spiral should turn clockwise or not [true].

Example

```
(spiral turns:4 sites:80)
```

4.10.7 wedge

Defines a triangular wedge shaped graph, with one vertex at the top and three vertices along the bottom.

Format

```
(wedge <dimFunction> [<dimFunction>])
```

where:

- `<dimFunction>`: Number of rows.

- `[<dimFunction>]`: Number of columns.

Example

```
(wedge 3)
```

Remarks

Wedges can be used to add triangular arms to Alquerque boards.

4.11 Operators

This section contains the operations that can be performed on graphs that describe game boards. These typically involve transforming the graph, modifying it, or merging multiple sub-graphs.

4.11.1 add

Adds elements to a graph.

Format

```
(add [<graphFunction>] [vertices:{{<floatFunction>}}]
     ([edges:{{{<floatFunction>}}}] | [edges:{{<dimFunction>}}])
     [edgesCurved:{{{<floatFunction>}}}] ([cells:{{{<floatFunction>}}}]
     | [cells:{{<dimFunction>}}]) [connect:<boolean>])
```

where:

- [<graphFunction>]: The graph to remove elements from.
- [vertices:{{<floatFunction>}}]: Locations of vertices to add.
- [edges:{{{<floatFunction>}}}]: Indices of end point vertices to add.
- [edges:{{<dimFunction>}}]: Indices of end point vertices to add.
- [edgesCurved:{{{<floatFunction>}}}]: Locations of end points and tangents of edges to add.
- [cells:{{{<floatFunction>}}}]: Indices of vertices of faces to add.
- [cells:{{<dimFunction>}}]: Indices of vertices of faces to add.
- [connect:<boolean>]: Whether to connect newly added vertices to nearby neighbours [false].

Examples

```
(add (square 4) vertices:{ { 1 2} })

(add edges:{ { { 0 0} { 1 1 } } })

(add (square 4) cells:{ { { 1 1} { 1 2 } { 3 2 } { 3 1 } } })

(add (square 2) edgesCurved:{ { { 0 0} { 1 0 } { 1 2 } { -1 2 } } })
```

Remarks

The elements to be added can be vertices, edges or faces. Edges and faces will create the specified vertices if they don't already exist. When defining curved edges, the first and second set of numbers are the end points locations and the third and fourth set of numbers are the tangent directions for the edge end points.

4.11.2 clip

Returns the result of clipping a graph to a specified shape.

Format

```
(clip <graphFunction> <poly>)
```

where:

- `<graphFunction>`: Graph to clip.
- `<poly>`: Float points defining clip region.

Example

```
(clip (square 4) (poly { { 1 1} { 1 3 } { 4 0 } }))
```

4.11.3 complete

Creates an edge between each pair of vertices in the graph.

Format

```
(complete <graphFunction> [eachCell:<boolean>])
```

where:

- `<graphFunction>`: The graph to complete.
- `[eachCell:<boolean>]`: Whether to complete each cell individually.

Examples

```
(complete (hex 1))

(complete (hex 3) eachCell:true)
```

4.11.4 dual

Returns the weak dual of the specified graph.

Format

```
(dual <graphFunction>)
```

where:

- <graphFunction>: The graph to take the weak dual of.

Example

```
(dual (square 5))
```

Remarks

The weak dual of a graph is obtained by creating a vertex at the midpoint of each face, then connecting with an edge vertices corresponding to adjacent faces. This is equivalent to the dual of the graph without the single "outside" vertex. The weak dual is non-transitive and always produces a smaller graph; applying (dual (dual ¡graph¿)) does not restore the original graph.

4.11.5 hole

Cuts a hole in a graph according to a specified shape.

Format

```
(hole <graphFunction> <poly>)
```

where:

- <graphFunction>: Graph to clip.
- <poly>: Float points defining hole region.

Example

```
(hole
    (square 8)
    (poly
        {
            {2.5 2.5} {2.5 5.5} {4.5 5.5} {4.5 4.5} {5.5 4.5} {5.5 2.5}
        }
    )
)
```

Remarks

Any face of the graph whose midpoint falls within the hole is removed, as are any edges or vertices isolated as a result.

4.11.6 intersect

Returns the intersection of two or more graphs.

Format

For making the intersection of two graphs.

```
(intersect <graphFunction> <graphFunction>)
```

where:

- `<graphFunction>`: First graph to intersect.
- `<graphFunction>`: Second graph to intersect.

For making the intersection of many graphs.

```
(intersect {<graphFunction>})
```

where:

- `{<graphFunction>}`: Graphs to intersect.

Examples

```
(intersect (square 4) (rectangle 2 5))

(intersect { (rectangle 6 2) (square 4) (rectangle 7 2) })
```

Remarks

The intersection of two or more graphs is composed of the vertices and edges that occur in all of those graphs.

4.11.7 keep

Keeps a specified shape within a graph and discards the remainder.

Format

```
(keep <graphFunction> <poly>)
```

where:

- `<graphFunction>`: Graph to modify.
- `<poly>`: Float points defining keep region.

Example

```
(keep (square 4) (poly { { 1 1} { 1 3 } { 4 0 } }))
```

4.11.8 layers

Makes multiple layers of the specified graph for 3D games.

Format

```
(layers <dimFunction> <graphFunction>)
```

where:

- `<dimFunction>`: Number of layers.
- `<graphFunction>`: The graph to layer.

Example

```
(layers 3 (square 3))
```

Remarks

The layers are stacked upon each one 1 unit apart. Layers are shown in isometric view from the side.

4.11.9 makeFaces

Recreates all possible non-overlapping faces for the given graph.

Format

```
(makeFaces <graphFunction>)
```

where:

- `<graphFunction>`: The graph to be modified.

Example

```
(makeFaces (square 5))
```

4.11.10 merge

Returns the result of merging two or more graphs.

Format

For making the merge of two graphs.

```
(merge <graphFunction> <graphFunction> [connect:<boolean>])
```

where:

- `<graphFunction>`: First graph to merge.
- `<graphFunction>`: Second graph to merge.
- `[connect:<boolean>]`: Whether to connect newly added vertices to nearby neighbours [false].

For making the merge of many graphs.

```
(merge {<graphFunction>} [connect:<boolean>])
```

where:

- `{<graphFunction>}`: Graphs to merge.

- [connect:<boolean>]: Whether to connect newly added vertices to nearby neighbours [false].

Examples

```
(merge (rectangle 6 2) (rectangle 3 5))

(merge { (rectangle 6 2) (square 4) (rectangle 7 2) })
```

Remarks

The graphs are overlaid with each other, such that incident vertices (i.e. those with the same location) are merged into a single vertex.

4.11.11 recoordinate

Regenerates the coordinate labels for the elements of a graph.

Format

```
(recoordinate [<siteType>] [<siteType>] [<siteType>] <graphFunction>)
```

where:

- [<siteType>]: First site type to recoordinate (Vertex/Edge/Cell).
- [<siteType>]: Second site type to recoordinate (Vertex/Edge/Cell).
- [<siteType>]: Third site type to recoordinate (Vertex/Edge/Cell).
- <graphFunction>: The graph whose vertices are to be renumbered.

Examples

```
(recoordinate (merge (rectangle 2 5) (square 5)))

(recoordinate Vertex (merge (rectangle 2 5) (square 5)))
```

4.11.12 remove

Removes elements from a graph.

Format

For removing some graph elements.

```
(remove <graphFunction> ([cells:{{{<float>}}}] |
     [cells:{<dimFunction>}]) ([edges:{{{<float>}}}]
     ([edges:{{<dimFunction>}}]) ([vertices:{{<float>}}]
     ([vertices:{<dimFunction>}]) [trimEdges:<boolean>])
```

where:

- <graphFunction>: The graph to remove elements from.
- [cells:{{{<float>}}}]: Indices of faces to remove.
- [cells:{<dimFunction>}]: Indices of faces to remove.
- [edges:{{{<float>}}}]: Indices of end points of edges to remove.
- [edges:{{<dimFunction>}}]: Indices of end points of edges to remove.
- [vertices:{{<float>}}]: Indices of vertices to remove.
- [vertices:{<dimFunction>}]: Indices of vertices to remove.
- [trimEdges:<boolean>]: Whether to trim edges orphaned by removing faces [true].

For removing some elements according to a polygon.

```
(remove <graphFunction> <poly> [trimEdges:<boolean>])
```

where:

- <graphFunction>: Graph to clip.
- <poly>: Float points defining hole region.
- [trimEdges:<boolean>]: Whether to trim edges orphaned by removing faces [true].

Examples

```
(remove (square 4) vertices:{ { 0.0 3.0} { 0.5 2 } })

(remove (square 4) cells:{0 1 2} edges:{ { 0 1 } { 1 2 } } vertices:{ 1 4 })

(remove
    (square 8)
    (poly
        {
            {2.5 2.5} {2.5 5.5} {4.5 5.5} {4.5 4.5} {5.5 4.5} {5.5 2.5}
        }
    )
)
```

Remarks

The elements to be removed can be vertices, edges or faces. Elements whose vertices can't be found will be ignored. Be careful when removing by index, as the graph is modified and renumbered with each removal. It is recommended to specify indices in decreasing order and to avoid removing vertices, edges and/or faces by index on the same call (instead, you can chain multiple removals by index together, one for each element type).

4.11.13 renumber

Renumbers the vertices of a graph into sequential order.

Format

```
(renumber [<siteType>] [<siteType>] [<siteType>] <graphFunction>)
```

where:

- [<siteType>]: First site type to renumber (Vertex/Edge/Cell).
- [<siteType>]: Second site type to renumber (Vertex/Edge/Cell).
- [<siteType>]: Third site type to renumber (Vertex/Edge/Cell).
- <graphFunction>: The graph whose vertices are to be renumbered.

Examples

```
(renumber (merge (rectangle 2 5) (square 5)))

(renumber Vertex (merge (rectangle 2 5) (square 5)))
```

Remarks

Vertices are renumbered from the lower left rightwards and upwards, in an upwards reading order. Renumbering can be useful after a union or merge operation combines different graphs.

4.11.14 rotate

Rotates a graph by the specified number of degrees anticlockwise.

Format

```
(rotate <floatFunction> <graphFunction>)
```

where:

- <floatFunction>: Number of degrees to rotate anticlockwise.

- <graphFunction>: The graph to rotate.

Example

```
(rotate 45 (square 5))
```

Remarks

The vertices within the graph are rotated about the graph's midpoint.

4.11.15 scale

Scales a graph by the specified amount.

Format

```
(scale <floatFunction> [<floatFunction>] [<floatFunction>]
     <graphFunction>)
```

where:

- <floatFunction>: Amount to scale in the x direction.
- [<floatFunction>]: Amount to scale in the y direction [scaleX].
- [<floatFunction>]: Amount to scale in the z direction [1].
- <graphFunction>: The graph to scale.

Examples

```
(scale 2 (square 5))

(scale 2 3.5 (square 5))
```

4.11.16 shift

Translate a graph by the specified x, y and z amounts.

Format

```
(shift <floatFunction> <floatFunction> [<floatFunction>]
     <graphFunction>)
```

where:

- <floatFunction>: Amount to translate in the x direction.
- <floatFunction>: Amount to translate in the y direction.
- [<floatFunction>]: Amount to translate in the z direction [0].
- <graphFunction>: The graph to rotate.

Example

```
(shift 0 10 (square 5))
```

Remarks

This operation modifies the locations of vertices within the graph.

4.11.17 skew

Skews a graph by the specified amount.

Format

```
(skew <float> <graphFunction>)
```

where:

- <float>: Amount to skew (1 gives a 45 degree skew).
- <graphFunction>: The graph to scale.

Example

```
(skew .5 (square 5))
```

4.11.18 splitCrossings

Splits edge crossings within a graph to create a new vertex at each crossing point.

Format

```
(splitCrossings <graphFunction>)
```

where:

- <graphFunction>: The graph to split edge crossings.

Example

```
(splitCrossings (merge (rectangle 2 5) (square 5)))
```

4.11.19 subdivide

Subdivides graph cells about their midpoint.

Format

```
(subdivide <graphFunction> [min:<dimFunction>])
```

where:

- <graphFunction>: The graph to subdivide.
- [min:<dimFunction>]: Minimum cell size to subdivide [1].

Example

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

Remarks

Each cell with N sides, where N ¿ min, will be split into N cells.

4.11.20 trim

Trims orphan vertices and edges from a graph.

Format

```
(trim <graphFunction>)
```

where:

- <graphFunction>: The graph to be trimmed.

Example

```
(trim (dual (square 5)))
```

Remarks

An orphan vertex is a vertex with no incident edge (note that pivot vertices are not removed). An orphan edge is an edge with an end point that has no incident edges apart from the edge itself.

4.11.21 union

Returns the union of two or more graphs.

Format

For making the union of two graphs.

```
(union <graphFunction> <graphFunction> [connect:<boolean>])
```

where:

- <graphFunction>: First graph to combine.
- <graphFunction>: Second graph to combine.
- [connect:<boolean>]: Whether to connect newly added vertices to nearby neighbours [false].

For making the union of many graphs.

```
(union {<graphFunction>} [connect:<boolean>])
```

where:

- {<graphFunction>}: Graphs to merge.
- [connect:<boolean>]: Whether to connect newly added vertices to nearby neighbours [false].

Examples

```
(union (square 5) (square 3))

(union { (rectangle 6 2) (square 4) (rectangle 7 2) })
```

Remarks

The graphs are simply combined with each other, with no connection between them.

5
Rules

Rule ludemes describe *how* the game is played. Games may be sub-divided into named *phases*, each with its own sub-rules, for clarity. Each games will typically have "start", "play" and "end" rules.

5.1 Rules

The *rules* ludeme describes the actual rules of play. These typically consist of "start", "play" and "end" rules.

5.1.1 rules

Sets the game's rules.

Format

For defining the rules with start, play and end.

```
(rules [<meta>] [<start>] <play> <end>)
```

where:

- [<meta>]: Metarules defined before play that supercede all other rules.
- [<start>]: Rules defining the starting position.
- <play>: Rules of play.
- <end>: Ending rules.

For defining the rules with some phases.

```
(rules [<meta>] [<start>] [<play>] phases:{<phase>} [<end>])
```

where:

- [<meta>]: Metarules defined before play that supercede all other rules.
- [<start>]: The starting rules.
- [<play>]: The playing rules shared between each phase.
- phases:{<phase>}: The phases of the game.
- [<end>]: The ending rules shared between each phase.

Examples

```
(rules
    (play (move Add (to (sites Empty))))
    (end (if (is Line 3) (result Mover Win)))
)

(rules
    (start (place "Ball" "Hand" count:3))
    phases:{
        (phase
            "Placement"
            (play (fromTo (from (handSite Mover)) (to (sites Empty))))
            (nextPhase ("HandEmpty" P2) "Movement")
        )
        (phase "Movement" (play (forEach Piece)))
    }
    (end (if (is Line 3) (result Mover Win)))
)
```

5.2 End

The `end` rules describe the terminating conditions of the game and the result of the game.

5.2.1 byScore

Is used to end a game based on the score of each player.

Format

```
(byScore [{<score>}])
```

where:

- `[{<score>}]`: The final score of each player.

Example

```
(byScore)
```

5.2.2 end

Defines the rules for ending a game.

Format

```
(end (<endRule> | {<endRule>}))
```

where:

- `<endRule>`: The ending rule.
- `{<endRule>}`: The ending rules.

Example

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

5.2.3 forEach

Applies the end condition to each player of a certain type.

Format

```
(forEach ([<roleType>] | [Track]) if:<boolean> <result>)
```

where:

- `[<roleType>]`: Role type to iterate through [Shared].
- `if:<boolean>`: Condition to apply.
- `<result>`: Result to return.

Example

```
(forEach NonMover if:(is Blocked Player) (result Player Loss))
```

5.2.4 if

Implements the condition(s) for ending the game, and deciding its result.

Format

```
(if <boolean> ([<if>] | [{<if>}]) [<result>])
```

where:

- `<boolean>`: Condition to end the game.
- `[<if>]`: Sub-condition to check.
- `[{<if>}]`: Sub-conditions to check.
- `[<result>]`: Default result to return if no sub-condition is satisfied.

Example

```
(if (is Mover (next)) (result Mover Win))
```

Remarks

If the stopping condition is met then this rule will return a result, whether any sub-conditions are defined or not.

5.2.5 result

Gives the result when an ending rule is reached for a specific player/team.

Format

```
(result <roleType> <resultType>)
```

where:

- <roleType>: The player or the team.
- <resultType>: The result type of the player or team.

Example

```
(result Mover Win)
```

5.3 Meta

The `meta` rules describe higher-level rules applied across the entire game.

5.3.1 automove

To apply automatically to the game all the legal moves only applicable to a single site.

Format

```
(automove [<boolean>])
```

where:

- `[<boolean>]`: The value of the auto move meta rule [true].

Example

```
(automove)
```

5.3.2 meta

Defines a metarule defined before play that supercedes all other rules.

Format

```
(meta ({<metaRule>} | <metaRule>))
```

where:

- `{<metaRule>}`: A collection of metarules.
- `<metaRule>`: A single metarule.

Example

```
(meta (swap))
```

5.3.3 noRepeat

Specifies a particular type of repetition that is forbidden in the game.

Format

```
(noRepeat [<repetitionType>])
```

where:

- `[<repetitionType>]`: Type of repetition to forbid [InGame].

Example

```
(noRepeat InTurn)
```

Remarks

The Infinite option disallows players from making consecutive sequences of moves that would lead to the same state twice, which would indicate the start of an infinite cycle of moves.

5.3.4 swap

To activate the swap rule.

Format

```
(swap [<boolean>])
```

where:

- `[<boolean>]`: The value of the auto move meta rule [true].

Example

```
(swap)
```

5.4 Phase

Games may be sub-divided into named *phases* for clarity. Each phase can contain its own sub-rules, which override the rules for the broader game while in that phase. Each phase can nominate a "next" phase to which control is relinquished under specified conditions.

5.4.1 nextPhase

Enables a player or all the players to proceed to another phase of the game.

Format

```
(nextPhase ([<roleType>] | [<player>]) [<boolean>] [<string>])
```

where:

- `[<roleType>]`: The roleType of the player [Shared].
- `[<player>]`: The index of the player.
- `[<boolean>]`: The condition to satisfy to go to another phase [true].
- `[<string>]`: The name of the phase.

Example

```
(nextPhase Mover (= (count Moves) 10) "Movement")
```

Remarks

If no phase is specified, moves to the next phase in the list, wrapping back to the first phase if needed. The ludeme returns Undefined (-1) if the condition is false or if the named phase does not exist.

5.4.2 phase

Defines the phase of a game.

Format

```
(phase <string> [<roleType>] [<mode>] [<play>] [<end>] ([<nextPhase>]
     | [{<nextPhase>}]))
```

where:

- `<string>`: The name of the phase.
- `[<roleType>]`: The roleType of the owner of the phase [Shared].

- [<mode>]: The mode of this phase within the game [mode defined for whole game)].
- [<play>]: The playing rules of this phase.
- [<end>]: The ending rules of this phase.
- [<nextPhase>]: The next phase of this phase.
- [{<nextPhase>}]: The next phases of this phase.

Example

```
(phase "Movement" (play (forEach Piece)))
```

Remarks

A phase can be defined for only one player.

5.5 Play

The `play` rules describe the actual rules of play, from the start to the end of each trial.

5.5.1 play

Checks the playing rules of the game.

Format

```
(play <moves>)
```

where:

- `<moves>`: The legal moves of the playing rules.

Example

```
(play (forEach Piece))
```

5.6 Start

The `start` rules describe the initial setup of equipment before play commences.

5.6.1 deal

To deal different components between players.

Format

```
(deal <dealableType> [int])
```

where:

- `<dealableType>`: Type of deal.
- `[int]`: The number of components to deal [1].

Example

```
(deal Dominoes 7)
```

5.6.2 start

Defines a starting position.

Format

```
(start ({<startRule>} | <startRule>))
```

where:

- {`<startRule>`}: The starting rules.
- `<startRule>`: The starting rule.

Example

```
(start
    {
        (place "Pawn1" {"F4" "F5" "F6" "F7" "F8" "F9" "G5" "G6" "G7" "G8"})
        (place "Knight1" {"F3" "G4" "G9" "F10"})
        (place "Pawn2" {"K4" "K5" "K6" "K7" "K8" "K9" "J5" "J6" "J7" "J8"})
        (place "Knight2" {"K3" "J4" "J9" "K10"})
    }
)
```

Remarks

For any game with starting rules, like pieces already placed on the board.

5.7 Start - DeductionPuzzle

Start rules specific to deduction puzzles typically involve setting hint values for puzzle challenges.

5.7.1 set

Sets a variable to a specified value in a deduction puzzle.

Format

```
(set [<siteType>] {{int}})
```

where:

- `[<siteType>]`: The graph element type [Cell].
- `{{int}}`: The first element of the pair is the index of the variable, the second one the value of the variable.

Example

```
(set
    {
        {1 9} {6 4} {11 8} {12 5} {16 1} {20 1} {25 6} {26 8} {30 1} {34 3}
        {40 4} {41 5} {42 7} {46 5} {50 7} {55 7} {58 9} {60 2} {65 3} {66 6}
        {72 8}
    }
)
```

Remarks

Applies to deduction puzzles.

5.8 Start - Place

The `place` rules to initially place items into playing sites.

5.8.1 place

Sets some aspect of the initial game state.

Format

For placing an item to a site.

```
(place <string> [<string>] [<siteType>] [<int>] [coord:<string>]
     [count:int] [state:int] [rotation:int] [invisibleTo:{<roleType>}]
     [maskedTo:{<roleType>}])
```

where:

- `<string>`: The name of the item.
- `[<string>]`: The name of the container.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<int>]`: The location to place a piece.
- `[coord:<string>]`: The coordinate of the location to place a piece.
- `[count:int]`: The number of the same piece to place [1].
- `[state:int]`: The local state value of the piece to place [Off].
- `[rotation:int]`: The rotation value of the piece to place [Off].
- `[invisibleTo:{<roleType>}]`: The list of the players where these locations will be invisible.
- `[maskedTo:{<roleType>}]`: The list of the players where these locations will be masked.

For placing item(s) to sites.

```
(place <string> [<siteType>] [{<int>}] [<region>]
     [{<string>}] [counts:{int}] [state:int] [rotation:int]
     [invisibleTo:{<roleType>}])
```

where:

- `<string>`: The item to place.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[{<int>}]`: The sites to fill.
- `[<region>]`: The region to fill.

- [{<string>}]: The coordinates of the sites to fill.
- [counts:{int}]: The number of pieces on the state.
- [state:int]: The local state value to put on each site.
- [rotation:int]: The rotation value to put on each site.
- [invisibleTo:{<roleType>}]: The list of the players where these locations will be invisible.

For placing items into a stack.

```
(place Stack (<string> | items:{<string>}) [<string>]
     [<siteType>] ([<int>] | [{<int>}] | [<region>] |
     [coord:<string>] | [{<string>}]) ([count:int] ([counts:{int}])
     [state:int] [rotation:int] [invisibleTo:{<roleType>}]
     [maskedTo:{<roleType>}])
```

where:

- <string>: The item to place on the stack.
- items:{<string>}: The name of the items on the stack to place.
- [<string>]: The name of the container.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: The location to place the stack.
- [{<int>}]: The locations to place the stacks.
- [<region>]: The region to place the stacks.
- [coord:<string>]: The coordinate of the location to place the stack.
- [{<string>}]: The coordinates of the sites to place the stacks.
- [count:int]: The number of the same piece to place on the stack [1].
- [counts:{int}]: The number of pieces on the stack.
- [state:int]: The local state value of the piece on the stack to place [Off].
- [rotation:int]: The rotation value of the piece on the stack to place [Off].
- [invisibleTo:{<roleType>}]: The list of the players where these locations will be invisible.
- [maskedTo:{<roleType>}]: The list of the players where these locations will be masked.

For placing randomly pieces.

```
(place Random [<int>] {<string>} [count:int] [invisibleTo:{<roleType>}]
     [maskedTo:{<roleType>}] [<siteType>])
```

where:

- [<int>]: The container in which to randomly place piece(s).
- {<string>}: The names of the item to place.
- [count:int]: The number of items to place [1].
- [invisibleTo:{<roleType>}]: The list of the players where these locations will be invisible.
- [maskedTo:{<roleType>}]: The list of the players where these locations will be masked.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].

For placing randomly a stack.

```
(place Random {<string>} [count:{<int>}] <int>
     [invisibleTo:{<roleType>}] [maskedTo:{<roleType>}] [<siteType>])
```

where:

- {<string>}: The names of each type of piece in the stack.
- [count:{<int>}]: The number of pieces of each piece in the stack.
- <int>: The site on which to place the stack.
- [invisibleTo:{<roleType>}]: The list of the players where these locations will be invisible.
- [maskedTo:{<roleType>}]: The list of the players where these locations will be masked.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].

For placing randomly a stack with specific number of each type of pieces.

```
(place Random {<count>} <int> [invisibleTo:{<roleType>}]
     [maskedTo:{<roleType>}] [<siteType>])
```

where:

- {<count>}: The items to be placed, with counts.
- <int>: The site on which to place the stack.
- [invisibleTo:{<roleType>}]: The list of the players where these locations will be invisible.
- [maskedTo:{<roleType>}]: The list of the players where these locations will be masked.

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].

Examples

```
(place "Pawn1" 0)

(place "Pawn1" (sites Bottom))

(place Stack items:{"Counter2" "Counter1"} 0)

(place Stack "Disc1" coord:"A1" count:5)

(place Random {"Pawn1" "Pawn2"})

(place Random {"Ball1"} count:29)

(place
    Random
    {
        (count "Pawn1" 8)
        (count "Rook1" 2)
        (count "Knight1" 2)
        (count "Bishop1" 2)
        (count "Queen1" 1)
        (count "King1" 1)
    }
    (handSite 1)
    maskedTo:{ P1 P2}
)
```

5.9 Start - Set

The `(set ...)` start 'super' ludeme sets some aspect of the initial game state. This can include initial scores for players, initial teams, starting amounts, etc.

5.9.1 set

Sets some aspect of the initial game state.

Format

For setting a site to a player.

```
(set <roleType> [<siteType>] [<int>] [coord:<string>]
     [invisibleTo:{<roleType>}] [maskedTo:{<roleType>}])
```

where:

- `<roleType>`: The owner of the site.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<int>]`: The location to place a piece.
- `[coord:<string>]`: The coordinate of the location to place a piece.
- `[invisibleTo:{<roleType>}]`: The list of the players where these locations will be invisible.
- `[maskedTo:{<roleType>}]`: The list of the players where these locations will be masked.

For setting sites to a player.

```
(set <roleType> [<siteType>] [{<int>}] [<region>] [{<string>}]
     [invisibleTo:{<roleType>}])
```

where:

- `<roleType>`: The owner of the site.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[{<int>}]`: The sites to fill.
- `[<region>]`: The region to fill.
- `[{<string>}]`: The coordinates of the sites to fill.
- `[invisibleTo:{<roleType>}]`: The list of the players where these locations will be invisible.

For setting the count, the cost or the phase to sites.

```
(set <setStartSitesType> int [<siteType>] (at:<int> | to:<region>))
```

where:

- `<setStartSitesType>`: The property to set.
- `int`: The value.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The site to set.
- `to:<region>`: The region to set.

For setting the amount or the score of a player.

```
(set <setStartPlayerType> [<roleType>] <int>)
```

where:

- `<setStartPlayerType>`: The property to set.
- `[<roleType>]`: The roleType of the player.
- `<int>`: The value to set.

For setting a team.

```
(set Team <int> {<roleType>})
```

where:

- `<int>`: The value to set.
- `{<roleType>}`: The roleType of the player.

For setting all the sites to be invisible.

```
(set AllInvisible [<siteType>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

Examples

```
(set P1 Vertex 5)

(set P1 Vertex (sites {0 5 6}))

(set Count 5 to:(sites Track))

(set Cost 5 Vertex at:10)

(set Phase 1 Cell at:3)

(set Amount 5000)

(set Team 1 { P1 P3})

(set AllInvisible)
```

5.9.2 setStartPlayerType

Defines the player properties that can be set in the starting rules.

Value	Description
Amount	Sets the initial amount for a player.
Score	Sets the intial score of a player.

5.9.3 setStartSitesType

Defines the properties of board sites that can be set in the starting rules.

Value	Description
Count	Sets the count of a site or region.
Cost	Sets the cost of a site or region.
Phase	Sets the phase of a site or region.

5.10 Start - Split

The `(split ...)` start 'super' ludeme to split objects between players.

5.10.1 split

Splits a deck of cards.

Format

```
(split Deck)
```

Example

```
(split Deck)
```

Remarks

This ludeme is used for card games.

6
Moves

Move ludemes define the legal moves for a given game state. We distinguish between:

- *decision moves* that involve a choice by the player,
- *effect moves* that are applied as the result of a decision, and
- *move generators* that iterate over the playable sites.

6.1 Decision

To specify that a move is a decision move.

6.1.1 move

Defines a decision move.

Format

For deciding to remove components.

`(move Remove [<siteType>] (<int> | <region>) [at:<whenType>] [<then>])`

where:

- `[<siteType>]`: The graph element type of the location [Cell (or Vertex if the main board uses this)].
- `<int>`: The location to remove a piece.
- `<region>`: The locations to remove a piece.
- `[at:<whenType>]`: When to perform the removal [immediately].
- `[<then>]`: The moves applied after that move is applied.

For deciding the trump suit of a card game.

`(move Set TrumpSuit (<int> | <difference>) [<then>])`

where:

- `<int>`: The suit to choose.
- `<difference>`: The possible suits to choose.
- `[<then>]`: The moves applied after that move is applied.

For deciding the next player.

`(move Set NextPlayer (<player> | <intArrayFunction>) [<then>])`

where:

- `<player>`: The data of the next player.
- `<intArrayFunction>`: The indices of the next players.
- `[<then>]`: The moves applied after that move is applied.

For deciding to set the direction.

```
(move Set Direction [<to>] ([{<int>}] | [<int>]) [previous:<boolean>]
     [next:<boolean>] [<then>])
```

where:

- [<to>]: Description of the “to” location [(to (from))].
- [{<int>}]: The index of the possible new directions.
- [<int>]: The index of the possible new direction.
- [previous:<boolean>]: True to allow movement to the left [true].
- [next:<boolean>]: True to allow movement to the right [true].
- [<then>]: The moves applied after that move is applied.

For deciding to step.

```
(move Step [<from>] [<direction>] <to> [stack:<boolean>] [<then>])
```

where:

- [<from>]: Description of “from” location [(from)].
- [<direction>]: The directions of the move [Adjacent].
- <to>: Description of the “to” location.
- [stack:<boolean>]: True if the move is applied to a stack [false].
- [<then>]: Moves to apply after this one.

For deciding to slide.

```
(move Slide [<from>] [<string>] [<direction>] [<between>] [<to>]
     [<then>])
```

where:

- [<from>]: Description of the “from” location [(from)].
- [<string>]: The track on which to slide.
- [<direction>]: The directions of the move [Adjacent].
- [<between>]: Description of the location(s) between “from” and “to”.
- [<to>]: Description of the “to” location [(to if:(is In (to) (sites Empty)))].
- [<then>]: Moves to apply after this one.

For deciding to shoot.

```
(move Shoot <piece> [<from>] [<absoluteDirection>] [<between>] [<to>]
     [<then>])
```

where:

- `<piece>`: The data about the piece to shoot.
- `[<from>]`: The “from” location [(lastTo)].
- `[<absoluteDirection>]`: The direction to follow [Adjacent].
- `[<between>]`: The location(s) between “from” and “to”.
- `[<to>]`: The condition on the “to” location to allow shooting [(to if:(in (to) (sites Empty)))].
- `[<then>]`: The moves applied after that move is applied.

For deciding to select sites.

```
(move Select <from> [<to>] [<then>])
```

where:

- `<from>`: Describes the “from” location to select [(from)].
- `[<to>]`: Describes the “to” location to select.
- `[<then>]`: The moves applied after that move is applied.

For deciding to vote or propose.

```
(move <moveMessageType> (<string> | {<string>}) [<then>])
```

where:

- `<moveMessageType>`: The type of move.
- `<string>`: The message.
- `{<string>}`: The messages.
- `[<then>]`: The moves applied after that move is applied.

For deciding to promote.

```
(move Promote [<siteType>] [<int>] <piece> ([<player>] | [<roleType>])
     [<then>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<int>]`: The location of the piece to promote [(to)].
- `<piece>`: The data about the promoted pieces.

- [<player>]: Data of the owner of the promoted piece.
- [<roleType>]: RoleType of the owner of the promoted piece.
- [<then>]: The moves applied after that move is applied.

For deciding to pass or play a card.

```
(move <moveSimpleType> [<then>])
```

where:

- <moveSimpleType>: The type of move.
- [<then>]: The moves applied after that move is applied.

For deciding to leap.

```
(move Leap [<from>] {{<stepType>}} [forward:<boolean>]
     [rotations:<boolean>] <to> [<then>])
```

where:

- [<from>]: The from location [(from)].
- {{<stepType>}}: The walk to follow.
- [forward:<boolean>]: True if the move can only move forward according to the direction of the piece [false].
- [rotations:<boolean>]: True if the move includes all the rotations of the walk [true].
- <to>: The data about the location to move.
- [<then>]: The moves applied after that move is applied.

For deciding to hop.

```
(move Hop [<from>] [<direction>] [<between>] <to> [stack:<boolean>]
     [<then>])
```

where:

- [<from>]: The data of the from location [(from)].
- [<direction>]: The directions of the move [Adjacent].
- [<between>]: The information about the locations between “from” and “to” [(between if:true)].
- <to>: The condition on the location to move.
- [stack:<boolean>]: True if the move has to be applied for stack [false].
- [<then>]: The moves applied after that move is applied.

For deciding to move a piece.

```
(move <from> <to> [count:<int>] [copy:<boolean>] [stack:<boolean>]
     [<roleType>] [<then>])
```

where:

- <from>: The data of the “from” location [(from)].
- <to>: The data of the “to” location.
- [count:<int>]: The number of pieces to move.
- [copy:<boolean>]: Whether to duplicate the piece rather than moving it [false].
- [stack:<boolean>]: To move a complete stack [false].
- [<roleType>]: The mover of the move.
- [<then>]: The moves applied after that move is applied.

For deciding to bet.

```
(move Bet (<player> | <roleType>) <rangeFunction> [<then>])
```

where:

- <player>: The data about the player to bet.
- <roleType>: The RoleType of the player to bet.
- <rangeFunction>: The range of the bet.
- [<then>]: The moves applied after that move is applied.

For deciding to add a piece or claim a site.

```
(move <moveSiteType> [<piece>] <to> [count:<int>] [stack:<boolean>]
     [<then>])
```

where:

- <moveSiteType>: The type of move.
- [<piece>]: The data about the components to add.
- <to>: The data on the location to add.
- [count:<int>]: The number of components to add [1].
- [stack:<boolean>]: True if the move has to be applied on a stack [false].
- [<then>]: The moves applied after that move is applied.

Examples

```
(move Remove (last To))

(move Set TrumpSuit (card Suit at:(handSite Shared)))

(move Set NextPlayer (player (mover)))

(move Set Direction)

(move Set Direction (to (last To)) next:false)

(move Step (to if:(is Empty (to))))

(move Step Forward (to if:(is Empty (to))))

(move
    Step
    (directions { FR FL })
    (to
        if:(or (is Empty (to)) (is Enemy (who at:(to))))
        (apply (remove (to)))
    )
)

(move Slide)

(move Slide Orthogonal)

(move
    Slide
    "AllTracks"
    (between if:(or (= (between) (from)) (is In (between) (sites Empty))))
    (to if:(is Enemy (who at:(to))) (apply (remove (to))))
    (then (set Counter))
)

(move Shoot (piece "Dot0"))

(move Select (from) (then (remove (last To))))

(move
    Select
    (from (sites Occupied by:Mover) if:(!= (state at:(to)) 0))
    (to (sites Occupied by:Next) if:(!= (state at:(to)) 0))
    (then
        (set
            State
            at:(last To)
            (% (+ (state at:(last From)) (state at:(last To))) 5)
        )
    )
)

(move Propose "End")

(move Vote "End")

(move Promote (last To) (piece {"Queen" "Knight" "Bishop" "Rook"}) Mover)

(move Pass)
```

6.1.2 moveMessageType

Defines the types of decision move relative to a message.

Value	Description
Propose	Makes a propose move.
Vote	Makes a vote move.

6.1.3 moveSimpleType

Defines the types of decision move corresponding to move with no parameters except the subsequents.

Value	Description
Pass	Makes a pass move.
PlayCard	Plays a card.

6.1.4 moveSiteType

Defines the types of decision move corresponding to a single site.

Value	Description
Add	Makes a add move.
Claim	Makes a claim move.

6.2 NonDecision - Effect

Effect moves are those moves that are applied as the result of a player decision.

6.2.1 add

Places one or more component(s) at a collection of sites or at one specific site.

Format

```
(add [<piece>] <to> [count:<int>] [stack:<boolean>] [<then>])
```

where:

- `[<piece>]`: The data about the components to add.
- `<to>`: The data on the location to add.
- `[count:<int>]`: The number of components to add [1].
- `[stack:<boolean>]`: True if the move has to be applied on a stack [false].
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(add (to (sites Empty)))

(add (to Cell (sites Empty Cell)))

(add (piece "Disc0") (to (last From)))

(add (piece "Disc0") (to (sites Empty)) (then (attract)))
```

Remarks

The "to" location is not updated until the move is made.

6.2.2 apply

Returns the effect to apply only if the condition is satisfied.

Format

```
(apply [if:<boolean>] [<nonDecision>])
```

where:

- `[if:<boolean>]`: The condition to satisfy to get the effect moves.

- [<nonDecision>]: The moves to apply to make the effect.

Example

```
(apply if:(is Mover P1) (moveAgain))
```

6.2.3 attract

Is used to attract all the pieces as close as possible to a site.

Format

```
(attract [<from>] [<absoluteDirection>] [<then>])
```

where:

- [<from>]: The data of the from location [(from)].
- [<absoluteDirection>]: The specific direction [Adjacent].
- [<then>]: The moves applied after that move is applied.

Example

```
(attract (from (last To)) Diagonal)
```

6.2.4 bet

Is used to bet an amount.

Format

```
(bet (<player> | <roleType>) <rangeFunction> [<then>])
```

where:

- <player>: The data about the player to bet.
- <roleType>: The roleType of the player to bet.
- <rangeFunction>: The range of the bet.
- [<then>]: The moves applied after that move is applied.

Example

```
(bet P1 (range 0 5))
```

Remarks

For games like Morra.

6.2.5 claim

Claims a site by adding a piece of the specified colour there.

Format

```
(claim [<piece>] <to> [<then>])
```

where:

- `[<piece>]`: The data about the components to claim.
- `<to>`: The data on the location to claim.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(claim (to Cell (site)) (then (and (addScore Mover 1) (moveAgain))))
```

Remarks

This ludeme is used for graph games.

6.2.6 custodial

Is used to apply an effect to all the sites flanked between two sites.

Format

```
(custodial [<from>] [<absoluteDirection>] [<between>] [<to>] [<then>])
```

where:

- `[<from>]`: The data about the sites used as an origin to flank [(from)].
- `[<absoluteDirection>]`: The direction to compute the flanking [Adjacent].
- `[<between>]`: The condition and effect on the pieces flanked [(between if:(is Enemy

(between)) (apply (remove (between))))].

- [<to>]: The condition on the pieces surrounding [(to if:(is Friend (to)))].
- [<then>]: The moves applied after that move is applied.

Example

```
(custodial
    (from (last To))
    Orthogonal
    (between
        (max 1)
        if:(is Enemy (who at:(between)))
        (apply (remove (between)))
    )
    (to if:(is Friend (who at:(to))))
)
```

Remarks

Used for example in all the Tafl games.

6.2.7 deal

Deals cards or dominoes to each player.

Format

```
(deal <dealableType> [<int>] [beginWith:<int>] [<then>])
```

where:

- <dealableType>: Type of deal.
- [<int>]: The number of components to deal [1].
- [beginWith:<int>]: To start to deal with a specific player.
- [<then>]: The moves applied after that move is applied.

Example

```
(deal Cards 3 beginWith:(mover))
```

6.2.8 directionCapture

Is used to capture all the pieces in the same direction of the location specified.

Format

```
(directionCapture [<from>] [<to>] [opposite:<boolean>] [<then>])
```

where:

- `[<from>]`: The origin of the move [(from)].
- `[<to>]`: The condition of the location to go [(to if:(is Enemy (to)) (apply (remove (from))))].
- `[opposite:<boolean>]`: True to capture in the opposite direction [false].
- `[<then>]`: The moves applied after that move is applied.

Example

```
(directionCapture
    (from (last To))
    (to if:(is Enemy (who at:(to))) (apply (remove (to))))
)
```

Remarks

For example, used in Fanorona.

6.2.9 enclose

Applies a move to an enclosed group.

Format

```
(enclose [<siteType>] [<from>] [<absoluteDirection>] [<between>]
     [<then>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<from>]`: The origin of the enclosed group [(from)].
- `[<absoluteDirection>]`: The direction to compute the group [Adjacent].
- `[<between>]`: The condition and effect on the pieces enclosed [(between if:(is Enemy (between)) (apply (remove (between))))].
- `[<then>]`: The moves applied after that move is applied.

Example

```
(enclose
    (from (last To))
    Orthogonal
    (between if:(is Enemy (who at:(between))) (apply (remove (between))))
)
```

Remarks

A group of components is 'enclosed' if it has no adjacent empty sites, where board sides count as boundaries. This ludeme is used for surround capture games such as Go.

6.2.10 flip

Is used to flip a piece.

Format

```
(flip [<siteType>] [<int>] [<then>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: The location to flip the piece [(to)].
- [<then>]: The moves applied after that move is applied.

Examples

```
(flip)

(flip (last To))
```

6.2.11 fromTo

Moves a piece from one site to another, possibly in another container, with no direction link between the "from" and "to" sites.

Format

```
(fromTo <from> <to> [count:<int>] [copy:<boolean>] [stack:<boolean>]
     [<roleType>] [<then>])
```

where:

- <from>: The data of the "from" location [(from)].
- <to>: The data of the "to" location.
- [count:<int>]: The number of pieces to move.
- [copy:<boolean>]: Whether to duplicate the piece rather than moving it [false].
- [stack:<boolean>]: To move a complete stack [false].
- [<roleType>]: The mover of the move.
- [<then>]: The moves applied after that move is applied.

Examples

```
(fromTo (from (last To)) (to (last From)))

(fromTo (from (handSite Mover)) (to (sites Empty)))

(fromTo (from (to)) (to (sites Empty)) count:(count at:(to)))

(fromTo (from (handSite Shared)) (to (sites Empty)) copy:true)
```

Remarks

If the "copy" parameter is set, then a copy of the piece is duplicated at the "to" site rather than actually moving there.

6.2.12 hop

Defines a hop in which a piece hops over a hurdle (the *pivot*) in a direction.

Format

```
(hop [<from>] [<direction>] [<between>] <to> [stack:<boolean>]
     [<then>])
```

where:

- [<from>]: The data of the from location [(from)].
- [<direction>]: The directions of the move [Adjacent].
- [<between>]: The information about the locations between "from" and "to" [(between if:true)].
- <to>: The condition on the location to move.
- [stack:<boolean>]: True if the move has to be applied for stack [false].
- [<then>]: The moves applied after that move is applied.

Examples

```
(hop
    (between if:(is Enemy (who at:(between))) (apply (remove (between))))
    (to if:(is Empty (to)))
)

(hop
    Orthogonal
    (between if:(is Friend (who at:(between))) (apply (remove (between))))
    (to if:(is Empty (to)))
)
```

Remarks

Capture moves in Draughts are typical hop moves. Note that we extend the standard definition of “hop” to include cases where the pivot is empty, for example in games such as Lines of Action and Quantum Leap.

6.2.13 intervene

Is used to apply an effect to all the sites flanking a site.

Format

```
(intervene [<from>] [<absoluteDirection>] [<between>] [<to>] [<then>])
```

where:

- [<from>]: The data about the sites to intervene [(from)].
- [<absoluteDirection>]: The direction to compute the flanking [Adjacent].
- [<between>]: The condition on the pieces flanked [(between (exact 1))].
- [<to>]: The condition and effect on the pieces flanking [(to if:(is Enemy (who at:(to))) (apply (remove (to))))].
- [<then>]: The moves applied after that move is applied.

Example

```
(intervene
    (from (last To))
    (to if:(is Enemy (who at:(to))) (apply (remove (to))))
)
```

6.2.14 leap

Allows a player to leap a piece to sites defined by walks through the board graph.

Format

```
(leap [<from>] {{<stepType>}} [forward:<boolean>]
     [rotations:<boolean>] <to> [<then>])
```

where:

- [<from>]: The site to leap from [(from)].
- {{<stepType>}}: The walk that defines the landing site(s).
- [forward:<boolean>]: Whether to only keep moves that land forwards [false].
- [rotations:<boolean>]: Whether to apply the leap to all rotations [true].
- <to>: Details about the site to move to.
- [<then>]: Moves to apply after the leap.

Example

```
(leap
    { { F F R F } { F F L F } }
    (to
        if:(or (is Empty (to)) (is Enemy (who at:(to))))
        (apply (if (is Enemy (who at:(to))) (remove (to))))
    )
)
```

Remarks

Use this ludeme to make leaping moves to pre-defined destination sites that do not care about intervening pieces, such as knight moves in Chess.

6.2.15 note

Makes a note to a player or to all the players.

Format

```
(note ([<int>] | [<roleType>]) <string> ([to:<player>] |
     [to:<roleType>]))
```

where:

- [<int>]: The index of the player to add at the beginning of the message.
- [<roleType>]: The role of the player to add at the beginning of the message.

- `<string>`: The message.
- `[to:<player>]`: The role of the player [ALL].
- `[to:<roleType>]`: The role of the player [ALL].

Example

```
(note "Opponent has played")
```

6.2.16 pass

Passes this turn.

Format

```
(pass [<then>])
```

where:

- `[<then>]`: The moves applied after that move is applied.

Example

```
(pass)
```

6.2.17 playCard

Plays any card in a player's hand to the board at their position.

Format

```
(playCard [<then>])
```

where:

- `[<then>]`: The moves applied after that move is applied.

Example

```
(playCard)
```

6.2.18 promote

Is used for promotion into another item.

Format

```
(promote [<siteType>] [<int>] <piece> ([<player>] | [<roleType>])
     [<then>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<int>]`: The location of the piece to promote [(to)].
- `<piece>`: The data about the promoted pieces.
- `[<player>]`: Index of the owner of the promoted piece.
- `[<roleType>]`: RoleType of the owner of the promoted piece.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(promote (last To) (piece {"Queen" "Knight" "Bishop" "Rook"}) Mover)
```

6.2.19 propose

Is used to propose something to the other players.

Format

```
(propose (<string> | {<string>}) [<then>])
```

where:

- `<string>`: The proposition.
- `{<string>}`: The propositions.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(propose "End")
```

6.2.20 push

Pushes all the pieces from a site in one direction.

Format

```
(push [<from>] <absoluteDirection> [<then>])
```

where:

- `[<from>]`: Description of the "from" location [(from)].
- `<absoluteDirection>`: The direction to push.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(push (from (last To)) E)
```

6.2.21 remove

Removes an item from a site.

Format

```
(remove [<siteType>] (<int> | <region>) [at:<whenType>] [<then>])
```

where:

- `[<siteType>]`: The graph element type of the location [Cell (or Vertex if the main board uses this)].
- `<int>`: The location to remove a piece.
- `<region>`: The locations to remove a piece.
- `[at:<whenType>]`: When to perform the removal [immediately].
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(remove (last To))

(remove (last To) at:EndOfTurn)
```

Remarks

If the site is empty, the move is not applied.

6.2.22 roll

Rolls the dice.

Format

```
(roll [<then>])
```

where:

- `[<then>]`: The moves applied after that move is applied.

Example

```
(roll)
```

6.2.23 satisfy

Defines constraints applied at run-time for filtering legal puzzle moves.

Format

```
(satisfy (<boolean> | {<boolean>}))
```

where:

- `<boolean>`: The constraint of the puzzle.
- `{<boolean>}`: The constraints of the puzzle.

Example

```
(satisfy (all Different))
```

Remarks

This ludeme applies to deduction puzzles.

6.2.24 select

Selects either a site or a pair of "from" and "to" locations.

Format

```
(select <from> [<to>] [<then>])
```

where:

- `<from>`: Describes the "from" location to select [(from)].
- `[<to>]`: Describes the "to" location to select.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(select (from) (then (remove (last To))))

(select
    (from (sites Occupied by:Mover) if:(!= (state at:(to)) 0))
    (to (sites Occupied by:Next) if:(!= (state at:(to)) 0))
    (then
        (set
            State
            at:(last To)
            (% (+ (state at:(last From)) (state at:(last To))) 5)
        )
    )
)
```

Remarks

This ludeme is used to select one or two sites in order to apply a consequence to them. If the "to" location is not specified, then it is assumed to be the same as the 'from' location.

6.2.25 shoot

Is used to shoot an item from one site to another with a specific direction.

Format

```
(shoot <piece> [<from>] [<absoluteDirection>] [<between>] [<to>]
      [<then>])
```

where:

- `<piece>`: The data about the piece to shoot.
- `[<from>]`: The "from" location [(lastTo)].
- `[<absoluteDirection>]`: The direction to follow [Adjacent].
- `[<between>]`: The location(s) between "from" and "to".
- `[<to>]`: The condition on the "to" location to allow shooting [(to if:(in (to) (sites Empty)))].
- `[<then>]`: The moves applied after that move is applied.

Example

```
(shoot (piece "Dot0"))
```

Remarks

This ludeme is used for games including Amazons.

6.2.26 slide

Slides a piece in a direction through a number of sites.

Format

```
(slide [<from>] [<string>] [<direction>] [<between>] [<to>] [<then>])
```

where:

- `[<from>]`: Description of the "from" location [(from)].
- `[<string>]`: The track on which to slide.
- `[<direction>]`: The directions of the move [Adjacent].
- `[<between>]`: Description of the location(s) between "from" and "to".
- `[<to>]`: Description of the "to" location [(to if:(is In (to) (sites Empty)))].
- `[<then>]`: Moves to apply after this one.

Examples

```
(slide)

(slide Orthogonal)

(slide
    "AllTracks"
    (between if:(or (= (between) (from)) (is In (between) (sites Empty))))
    (to if:(is Enemy (who at:(to))) (apply (remove (to))))
    (then (set Counter))
)
```

Remarks

The rook in Chess is an example of a piece that slides in Orthogonal directions. Pieces can be constrained to slide along predefined tracks, e.g. see Surakarta. Note that we extend the standard definition of “slide” to allow pieces to slide through other pieces if a specific condition is given, but that pieces are assumed to slide through empty sites only by default.

6.2.27 sow

Sows counters by removing them from a site then placing them one-by-one at each consecutive site along a track.

Format

```
(sow [<siteType>] [<int>] [count:<int>] [<string>] [owner:<int>]
     [if:<boolean>] [apply:<nonDecision>] [includeSelf:<boolean>]
     [origin:<boolean>] [skipIf:<boolean>] [backtracking:<boolean>]
     [<then>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: The origin of the sowing [(lastTo)].
- [count:<int>]: The number of pieces to sow [(count (lastTo))].
- [<string>]: The name of the track to sow [The first track if it exists].
- [owner:<int>]: The owner of the track.
- [if:<boolean>]: The condition to capture some counters after sowing [true].
- [apply:<nonDecision>]: The move to apply if the condition is satisfied.
- [includeSelf:<boolean>]: True if the origin is included in the sowing [true].
- [origin:<boolean>]: True to place a counter in the origin at the start of sowing [false].

- [skipIf:<boolean>]: The condition to skip a hole during the sowing.
- [backtracking:<boolean>]: Whether to apply the capture backwards from the "to" site.
- [<then>]: The moves applied after that move is applied.

Example

```
(sow
    if:(and
        (is In (to) (sites Next))
        (or (= (count at:(to)) 2) (= (count at:(to)) 3))
    )
    apply:(fromTo (from (to)) (to (mapEntry (mover))) count:(count at:(to)))
    includeSelf:false
    backtracking:true
)
```

Remarks

Sowing is used in Mancala games. A track must be defined on the board in order for sowing to work.

6.2.28 step

Moves to a connected site.

Format

```
(step [<from>] [<direction>] <to> [stack:<boolean>] [<then>])
```

where:

- [<from>]: Description of "from" location [(from)].
- [<direction>]: The directions of the move [Adjacent].
- <to>: Description of the "to" location.
- [stack:<boolean>]: True if the move is applied to a stack [false].
- [<then>]: Moves to apply after this one.

Examples

```
(step (to if:(is Empty (to))))

(step Forward (to if:(is Empty (to))))

(step
    (directions { FR FL })
    (to
        if:(or (is Empty (to)) (is Enemy (who at:(to))))
        (apply (remove (to)))
    )
)
```

6.2.29 surround

Is used to apply an effect to all the sites surrounded in a specific direction.

Format

```
(surround [<from>] [<relationType>] [<between>] [<to>] [except:<int>]
     [with:<piece>] [<then>])
```

where:

- `[<from>]`: The origin to surround [(from)].
- `[<relationType>]`: The way to surround [Adjacent].
- `[<between>]`: The condition and effect on the pieces to surround [(between if:(is Enemy (to)) (apply (remove (to))))].
- `[<to>]`: The condition on the pieces surrounding [(to if:(isFriend (between)))].
- `[except:<int>]`: The number of exceptions allowed to apply the effect [0].
- `[with:<piece>]`: The piece which should at least be in the surrounded pieces.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(surround
    (from (last To))
    Orthogonal
    (between
        if:(is Friend (who at:(between)))
        (apply (trigger "Checkmate" (mover)))
    )
    (to if:(not (is In (to) (sites Empty))))
)
```

6.2.30 then

Defines the subsequents of a move, to be applied after the move.

Format

```
(then <nonDecision> [applyAfterAllMoves:<boolean>])
```

where:

- `<nonDecision>`: The moves to apply afterwards.
- `[applyAfterAllMoves:<boolean>]`: For simultaneous game, to apply the subsequents when all the moves of all the players are applied.

Example

```
(then (moveAgain))
```

Remarks

This is used to define subsequent moves by the same player in a turn after a move is made.

6.2.31 trigger

Sets the 'triggered' value for a player for a specific event.

Format

```
(trigger <string> (<int> | <roleType>) [<then>])
```

where:

- `<string>`: The event to trigger.
- `<int>`: The index of the player.
- `<roleType>`: The roleType of the player.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(trigger "FlagCaptured" Next)
```

6.2.32 vote

Is used to propose something to the other players.

Format

```
(vote (<string> | {<string>}) [<then>])
```

where:

- <string>: The vote.
- {<string>}: The votes.
- [<then>]: The moves applied after that move is applied.

Example

```
(vote "End")
```

6.3 NonDecision - Effect - Requirement

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

6.3.1 avoidStoredState

Filters the legal moves to avoid reaching a specific state.

Format

```
(avoidStoredState <moves> [<then>])
```

where:

- `<moves>`: The moves to filter.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(avoidStoredState (forEach Piece))
```

Remarks

Example: Arimaa.

6.3.2 do

Applies a sequence of moves in a specified order, according to given conditions.

Format

```
(do <moves> [next:<moves>] [ifAfterwards:<boolean>] [<then>])
```

where:

- `<moves>`: Moves to be applied first.
- `[next:<moves>]`: Follow-up moves computed after the first set of moves.
- `[ifAfterwards:<boolean>]`: Moves must satisfy this condition afterwards to be legal.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(do (roll) next:(if (!= (count Pips) 0) (forEach Piece)))

(do
    (fromTo
        (from (sites Occupied by:All container:(mover)))
        (to (sites LineOfPlay))
        (then (set Visible (last To)))
    )
    ifAfterwards:(is pipsMatch)
)
```

Remarks

Use ifAfterwards to filter out moves that do not satisfy some required condition after they are applied.

6.3.3 firstMoveOnTrack

Returns the first legal move on the track.

Format

```
(firstMoveOnTrack [<string>] [<roleType>] <moves> [<then>])
```

where:

- `[<string>]`: The name of the track.
- `[<roleType>]`: The owner of the track.
- `<moves>`: The moves to check.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(firstMoveOnTrack (forEach Piece))
```

Remarks

Example Backgammon.

6.3.4 priority

Returns the first list of moves with a non-empty set of moves.

Format

For selecting the first set of moves with a legal move between many set of moves.

```
(priority {<moves>} [<then>])
```

where:

- {<moves>}: The list of moves.
- [<then>]: The moves applied after that move is applied.

For selecting the first set of moves with a legal move between two moves.

```
(priority <moves> <moves> [<then>])
```

where:

- <moves>: The first set of moves.
- <moves>: The second set of moves.
- [<then>]: The moves applied after that move is applied.

Examples

```
(priority
    {
        (forEach Piece "Leopard" (step (to if:(is Enemy (who at:(to))))))
        (forEach Piece "Leopard" (step (to if:(is In (to) (sites Empty)))))
    }
)

(priority
    (forEach Piece "Leopard" (step (to if:(is Enemy (who at:(to))))))
    (forEach Piece "Leopard" (step (to if:(is In (to) (sites Empty)))))
)
```

Remarks

To prioritise a list of legal moves over another. For example in some draughts games, if you can capture, you must capture, if not you can move normally.

6.4 NonDecision - Effect - Requirement - Max

The `(max ...)` ‘super’ ludeme filters a list of moves to maximise a property.

6.4.1 max

Filters a list of legal moves to keep only the moves allowing the maximum number of moves in a turn.

Format

For getting the moves with the max captures or the max number of legal moves in the turn.

```
(max <maxMovesType> <moves> [<then>])
```

where:

- `<maxMovesType>`: The type of property to maximise.
- `<moves>`: The moves to filter.
- `[<then>]`: The moves applied after that move is applied.

For getting the moves with the max distance.

```
(max Distance [<string>] [<roleType>] <moves> [<then>])
```

where:

- `[<string>]`: The name of the track.
- `[<roleType>]`: The owner of the track.
- `<moves>`: The moves to filter.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(max Moves (forEach Piece))

(max Captures (forEach Piece))

(max Distance (forEach Piece))
```

6.4.2 maxMovesType

Defines the types of properties which can be used for the Max super ludeme with only a move ludeme in entry.

Value	Description
Moves	To ilter a list of legal moves to keep only the moves allowing the maximum number of moves in a turn.
Captures	To filter a list of moves to keep only the moves doing the maximum possible number of captures.

6.5 NonDecision - Effect - Set

The (set ...) 'super' ludeme sets some aspect of the game state in response to a move. This includes, for example, setting a counter value, or the next player, or the state of a site, etc.

6.5.1 set

Sets some aspect of the game state in response to a move.

Format

For setting the trump suit.

```
(set TrumpSuit (<int> | <difference>) [<then>])
```

where:

- <int>: The suit to choose.
- <difference>: The possible suits to choose.
- [<then>]: The moves applied after that move is applied.

For setting the next player.

```
(set NextPlayer (<player> | <intArrayFunction>) [<then>])
```

where:

- <player>: The data of the next player.
- <intArrayFunction>: The indices of the next players.
- [<then>]: The moves applied after that move is applied.

For setting the direction.

```
(set Direction [<to>] ([{<int>}] | [<int>]) [previous:<boolean>]
     [next:<boolean>] [<then>])
```

where:

- [<to>]: Description of the "to" location [(to (from))].
- [{<int>}]: The index of the possible new directions.
- [<int>]: The index of the possible new direction.
- [previous:<boolean>]: True to allow movement to the left [true].
- [next:<boolean>]: True to allow movement to the right [true].
- [<then>]: The moves applied after that move is applied.

For setting the value or the score of a player.

```
(set <setPlayerType> (<player> | <roleType>) <int> [<then>])
```

where:

- `<setPlayerType>`: The type of property to set.
- `<player>`: The index of the player.
- `<roleType>`: The role of the player.
- `<int>`: The value of the player.
- `[<then>]`: The moves applied after that move is applied.

For setting the pending values.

```
(set Pending ([<int>] | [<region>]) [<then>])
```

where:

- `[<int>]`: The value of the pending state [1].
- `[<region>]`: The set of locations to put in pending.
- `[<then>]`: The moves to apply afterwards.

For setting the counter or the variables.

```
(set <setValueType> [<int>] [<then>])
```

where:

- `<setValueType>`: The type of property to set.
- `[<int>]`: The new counter value [-1].
- `[<then>]`: The moves to apply afterwards.

For setting the count or the state of a site.

```
(set <setSiteType> [<siteType>] at:<int> <int> [<then>])
```

where:

- `<setSiteType>`: The type of property to set.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The site to set.
- `<int>`: The new value.
- `[<then>]`: The moves to apply afterwards.

For getting the hidden information of a site for a player.

```
(set <setRegionType> [<siteType>] (<int> | <region>) [level:<int>]
     ([<player>] | [<roleType>]) [stack:<boolean>] [<then>])
```

where:

- `<setRegionType>`: The type of property to set.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `<int>`: The site.
- `<region>`: The region.
- `[level:<int>]`: The level of the site to set [0].
- `[<player>]`: The index of the player for whom the site or region will be set.
- `[<roleType>]`: A role for which the site or region will be set.
- `[stack:<boolean>]`: True if the set has to be applied to a stack.
- `[<then>]`: The moves to apply afterwards.

Examples

```
(set TrumpSuit (card Suit at:(handSite Shared)))

(set NextPlayer (player (mover)))

(set Direction)

(set Direction (to (last To)) next:false)

(set Value Mover 1)

(set Score P1 50)

(set Pending)

(set Pending (sites From (forEach Piece)))

(set Counter -1)

(set Var (value Piece of:(last To)))

(set State at:(last To) (mover))

(set Count at:(last To) 10)

(set Masked (sites P2 "Home") P1)
```

6.5.2 setPlayerType

Defines properties related to the players that can be set in the game state.

Value	Description
Value	Sets the value associated with a player.
Score	Sets the score of a player.

6.5.3 setRegionType

Defines properties of player regions that can be set in the game state.

Value	Description
Visible	Makes specified sites visible to a given player.
Masked	Masks specified sites from a given player.
Invisible	Makes specified sites invisible to a given player.

6.5.4 setSiteType

Defines properties of sites that can be set in the game state.

Value	Description
Count	Set the count value for specified sites.
State	Set the local state value for specified sites.

6.5.5 setValueType

Defines the types of integer values that can be set in the game state.

Value	Description
Counter	Sets the counter of the game state.
Var	Sets the 'var' variable of the game state.
Pot	Sets the pot of the game state.

6.6 NonDecision - Effect - State

State move generators create moves based on certain properties of the game state.

6.6.1 addScore

Adds a value to the score of a player.

Format

For adding a score to a player.

```
(addScore (<player> | <roleType>) <int> [<then>])
```

where:

- <player>: The index of the player.
- <roleType>: The roleType of the player.
- <int>: The score of the player.
- [<then>]: The moves applied after that move is applied.

For adding a score to many players.

```
(addScore ({<int>} | {<roleType>}) {<int>} [<then>])
```

where:

- {<int>}: The indices of the players.
- {<roleType>}: The roleType of the players.
- {<int>}: The scores to add.
- [<then>]: The moves applied after that move is applied.

Examples

```
(addScore Mover 50)

(addScore { P1 P2} { 50 10 })
```

6.6.2 moveAgain

Is used to move again.

Format

```
(moveAgain)
```

Example

```
(moveAgain)
```

Remarks

For games with multiple moves in a turn.

6.6.3 rememberState

Stores the current state.

Format

```
(rememberState [<then>])
```

where:

- `[<then>]`: The moves applied after that move is applied.

Example

```
(rememberState)
```

6.7 NonDecision - Effect - State - Swap

The `(swap ...)` 'super' ludeme swaps two pieces or two players.

6.7.1 swap

Swaps two players or two pieces.

Format

For swapping two pieces.

```
(swap Pieces [<int>] [<int>] [<then>])
```

where:

- `[<int>]`: The first location [(lastFrom)].
- `[<int>]`: The second location [(lastTo)].
- `[<then>]`: The moves applied after that move is applied.

For swapping two players.

```
(swap Players (<int> | <roleType>) (<int> (<roleType>) [<then>])
```

where:

- `<int>`: The index of the first player.
- `<roleType>`: The role of the first player.
- `<int>`: The index of the second player.
- `<roleType>`: The role of the second player.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(swap Pieces)

(swap Players P1 P2)
```

6.8 NonDecision - Effect - Take

The (take ...) ‘super’ ludeme is used to take piece or the control of enemy pieces.

6.8.1 take

Takes a piece or the control of pieces.

Format

For taking a domino.

```
(take Domino [<then>])
```

where:

- [<then>]: The moves applied after that move is applied.

For taking the control of the pieces of another player.

```
(take Control (of:<roleType> | oF:<int>) (by:<roleType> (bY:<int>)
    [<siteType>] [<then>])
```

where:

- of:<roleType>: The player index of the pieces to take control of.
- oF:<int>: The player index of the pieces to take control of.
- by:<roleType>: The player index of the player taking control.
- bY:<int>: The player index of the player taking control.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<then>]: The moves applied after that move is applied.

Examples

```
(take Domino)

(take Control of:P1 by:Mover)
```

6.9 NonDecision - Operators - Foreach

Move generators are functions that iterate over playable sites and generate moves according to specified criteria.

6.9.1 forEach

Iterates over a set of items.

Format

For iterating the dice.

```
(forEach Die [<int>] [combined:<boolean>] [replayDouble:<boolean>]
     [if:<boolean>] <moves> [<then>])
```

where:

- `[<int>]`: The index of the dice container [0].
- `[combined:<boolean>]`: True if the combination is allowed [false].
- `[replayDouble:<boolean>]`: True if double allows a second move [false].
- `[if:<boolean>]`: The condition to satisfy to move [true].
- `<moves>`: The moves to apply.
- `[<then>]`: The moves applied after that move is applied.

For iterating on the directions.

```
(forEach Direction [<from>] [<direction>] [<between>] (<to> | <moves>)
     [<then>])
```

where:

- `[<from>]`: The origin of the movement [(from)].
- `[<direction>]`: The directions of the move [Adjacent].
- `[<between>]`: The data on the locations between the from location and the to location [(between (exact 1))].
- `<to>`: The data on the location to move.
- `<moves>`: The moves to applied on these directions.
- `[<then>]`: The moves applied after that move is applied.

For iterating on the sites of a region.

```
(forEach Site <region> <moves> [noMoveYet:<moves>] [<then>])
```

where:

- `<region>`: The region used.
- `<moves>`: The move to apply.
- `[noMoveYet:<moves>]`: The moves to apply if the list of moves resulting from the generator is empty.
- `[<then>]`: The moves applied after that move is applied.

For iterating on the pieces.

```
(forEach Piece ([<string>] | [{<string>}]) ([container:<int>]
     ([<string>]) [<moves>] ([<player>] | [<roleType>])
     [top:<boolean>] [<siteType>] [<then>])
```

where:

- `[<string>]`: The name of the piece.
- `[{<string>}]`: The names of the pieces.
- `[container:<int>]`: The index of the container.
- `[<string>]`: The name of the container.
- `[<moves>]`: The specific moves to apply to the pieces.
- `[<player>]`: The owner of the piece [(mover)].
- `[<roleType>]`: The role type of the owner of the piece [Mover].
- `[top:<boolean>]`: To apply the move only to the top piece in case of a stack [false].
- `[<siteType>]`: Type of graph element.
- `[<then>]`: The moves applied after that move is applied.

For iterating on the players.

```
(forEach Player ([container:<int>] | [<string>]) [<moves>] ([<player>]
     | [<roleType>]) [top:<boolean>] [<siteType>] [<then>])
```

where:

- `[container:<int>]`: The index of the container.
- `[<string>]`: The name of the container.
- `[<moves>]`: The specific moves to apply to the pieces.
- `[<player>]`: The owner of the piece [(mover)].
- `[<roleType>]`: The role type of the owner of the piece [Mover].
- `[top:<boolean>]`: To apply the move only to the top piece in case of a stack [false].

- [<siteType>]: Type of graph element.
- [<then>]: The moves applied after that move is applied.

Examples

```
(forEach
    Die
    (if
        (= (pips) 5)
        (or (forEach Piece "Pawn") (forEach Piece "King_noCross"))
        (if
            (= (pips) 4)
            (forEach Piece "Elephant")
            (if
                (= (pips) 3)
                (forEach Piece "Knight")
                (if (= (pips) 2) (forEach Piece "Boat"))
            )
        )
    )
)

(forEach
    Direction
    (from (to))
    (directions { FR FL })
    (to
        if:(or (is In (to) (sites Empty)) (is Enemy (who at:(to))))
        (apply
            (fromTo
                (from)
                (to
                    if:(or (is Empty (to)) (is Enemy (who at:(to))))
                    (apply (remove (to)))
                )
            )
        )
    )
)

(forEach
    Site
    (intersection (sites Around (last To)) (sites Occupied by:Next))
    (and (remove (site)) (add (piece (id "Ball" Mover)) (to (site))))
)

(forEach Piece)

(forEach Piece "Bear" (step (to if:(= (what at:(to)) (id "Seal1")))))

(forEach Player)
```

6.10 NonDecision - Operators - Logical

Logical move generators are used to combine or filter existing lists of moves.

6.10.1 allCombinations

Generates all combinations (i.e. the cross product) between two lists of moves.

Format

```
(allCombinations <moves> <moves> [<then>])
```

where:

- `<moves>`: The first list.
- `<moves>`: The second list.
- `[<then>]`: The moves applied after that move is applied.

Example

```
(allCombinations
    (add (piece (id "Disc0") state:(mover)) (to (site)))
    (flip (between))
)
```

6.10.2 and

Is used to combine lists of moves.

Format

For making a move between two sets of moves.

```
(and <moves> <moves> [<then>])
```

where:

- `<moves>`: The first move.
- `<moves>`: The second move.
- `[<then>]`: The moves applied after that move is applied.

For making a move between many sets of moves.

```
(and {<moves>} [<then>])
```

where:

- {<moves>}: The list of moves.
- [<then>]: The moves applied after that move is applied.

Examples

```
(and (set Score P1 100) (set Score P2 100))

(and { (set Score P1 100) (set Score P2 100) (set Score P3 100) })
```

6.10.3 append

Appends a list of moves to each move in a list.

Format

```
(append <nonDecision> [<then>])
```

where:

- <nonDecision>: The moves to merge.
- [<then>]: The moves applied after that move is applied.

Example

```
(append
    (custodial
        (between
            if:(is Enemy (state at:(between)))
            (apply
                (allCombinations
                    (add (piece "Disc0" state:(mover)) (to (site)))
                    (flip (between))
                )
            )
        )
        (to if:(is Friend (state at:(to))))
    )
    (then
        (and
            (set Score P1 (count Sites in:(sites State 1)))
            (set Score P2 (count Sites in:(sites State 2)))
        )
    )
)
```

6.10.4 if

Returns, depending on the condition, a list of legal moves or an other list.

Format

```
(if <boolean> <moves> [<moves>] [<then>])
```

where:

- `<boolean>`: The condition to satisfy to get the first list of legal moves.
- `<moves>`: The first list of legal moves.
- `[<moves>]`: The other list of legal moves if the condition is not satisfy.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(if (is Mover P1) (moveAgain))

(if (is Mover P1) (moveAgain) (remove (last To)))
```

6.10.5 or

Moves one of the moves in the list.

Format

For making a move between two sets of moves.

```
(or <moves> <moves> [<then>])
```

where:

- `<moves>`: The first move.
- `<moves>`: The second move.
- `[<then>]`: The moves applied after that move is applied.

For making a move between many sets of moves.

```
(or {<moves>} [<then>])
```

where:

- `{<moves>}`: The list of moves.
- `[<then>]`: The moves applied after that move is applied.

Examples

```
(or (set Score P1 100) (set Score P2 100))

(or { (set Score P1 100) (set Score P2 100) (set Score P3 100) })
```

7
Boolean Functions

Boolean functions are ludemes that return a "true" or "false" result to some query about the current game state. They verify the *existence* of a given condition in the current game state.

7.1 All

All is a 'super' ludeme that returns whether all aspects of a certain query about the game state are true, such as whether all players passed or all dice have been used.

7.1.1 all

Returns whether all aspects of the specified query are true.

Format

```
(all <allType>)
```

where:
- <allType>: The query type to perform.

Examples

```
(all DiceUsed)

(all Passed)
```

7.1.2 allType

Defines the query types that can be used for an `(all ...)` test.

Value	Description
DiceUsed	Returns whether all the dice have been used in the current turn.
DiceEqual	Returns whether all the dice are equal when they are rolled.
Passed	Returns whether all players have passed in succession.

7.2 Can

Can is a ‘super’ ludeme that returns whether a given property can be achieved in the current game state, such as whether the player can make at least one move.

7.2.1 can

Returns whether a given property can be achieved in the current game state.

Format

```
(can Move <moves>)
```

where:

- `<moves>`: List of moves.

Example

```
(can Move (forEach Piece))
```

7.3 DeductionPuzzle

Deduction puzzle queries return a boolean result based on whether certain constraints are respected in the current state of a puzzle challenge solution.

7.3.1 forAll

Returns true if the constraint is satisfied for each element.

Format

```
(forAll <puzzleElementType> <boolean>)
```

where:

- `<puzzleElementType>`: The type of the graph element.
- `<boolean>`: The constraint to check.

Example

```
(forAll Hint (is Count (sites Around (from) includeSelf:true) of:1 (hint)))
```

Remarks

This is used to test a constraint on each vertex, edge, face or site with a hint. This works only for deduction puzzles.

7.4 DeductionPuzzle - All

All is a 'super' puzzle ludeme that returns whether all aspects of a certain query about the puzzle state are true, such as whether all values in a region are different.

7.4.1 all

Whether the specified query is all true for a deduction puzzle.

Format

```
(all Different [<siteType>] [<region>] ([except:<int>] |
     [excepts:{<int>}]))
```

where:

- `[<siteType>]`: Type of graph elements to return [Cell].
- `[<region>]`: The region to check [Regions].
- `[except:<int>]`: The exception on the test.
- `[excepts:{<int>}]`: The exceptions on the test.

Examples

```
(all Different)

(all Different except:0)
```

7.5 DeductionPuzzle - Is

The `(is ...)` puzzle 'super' ludeme returns a true/false result to a given query about the puzzle state. The type of query is defined by a parameter specified by the user, and typically refer to constraints that the puzzle must satisfy, for example whether all values in a region are different, or sum to a certain hint value, etc.

7.5.1 is

Whether the specified query is true for a deduction puzzle.

Format

For solving a puzzle.

```
(is Solved)
```

For the unique constraint.

```
(is Unique [<siteType>])
```

where:

- `[<siteType>]`: The graph element type [Cell].

For a constraint related to count or sum.

```
(is <isPuzzleRegionResultType> [<siteType>] [<region>] [of:<int>]
    [<string>] <int>)
```

where:

- `<isPuzzleRegionResultType>`: The query type to perform.
- `[<siteType>]`: The graph element of the region [Cell].
- `[<region>]`: The region [Regions].
- `[of:<int>]`: The index of the piece [1].
- `[<string>]`: The name of the region to check.
- `<int>`: The result to check.

Examples

```
(is Solved)

(is Unique)

(is Count (sites All) of:1 8)

(is Sum 5)
```

7.5.2 isPuzzleRegionResultType

Defines the types of Is test for puzzle according to region and a specific result to check.

Value	Description
Count	To check if the count of a region is equal to the result.
Sum	To check if the sum of a region is equal to the result.

7.6 Is

The `(is ...)` 'super' ludeme returns whether a given query about the game state is true or not. Such queries might include whether a given piece belongs to a certain player, or is visible, whether certain regions are connected, etc.

7.6.1 is

Returns whether the specified query about the game state is true or not.

Format

For testing a tree.

```
(is <isTreeType> (<player> | <roleType>))
```

where:

- `<isTreeType>`: The type of query to perform.
- `<player>`: Data about the owner of the tree.
- `<roleType>`: RoleType of the owner of the tree.

For testing if a graph is regular.

```
(is RegularGraph (<player> | <roleType>) ([k:<int>] ([odd:<boolean>] |
    [even:<boolean>]))
```

where:

- `<player>`: The owner of the tree.
- `<roleType>`: RoleType of the owner of the tree.
- `[k:<int>]`: The parameter of k-regular graph.
- `[odd:<boolean>]`: Flag to recognize the k (in k-regular graph) is odd or not.
- `[even:<boolean>]`: Flag to recognize the k (in k-regular graph) is even or not.

For test relative to player.

```
(is <isPlayerType> (<int> | <roleType>))
```

where:

- `<isPlayerType>`: The type of query to perform.
- `<int>`: Index of the player or the component.
- `<roleType>`: The Role type corresponding to the index.

For a triggered test.

`(is Triggered <string> (<int> | <roleType>))`

where:

- `<string>`: The event triggered.
- `<int>`: Index of the player or the component.
- `<roleType>`: The Role type corresponding to the index.

For a test with no parameters.

`(is <isSimpleType>)`

where:

- `<isSimpleType>`: The type of query to perform.

For testing two edges crossing each other.

`(is Crossing <int> <int>)`

where:

- `<int>`: The index of the first edge.
- `<int>`: The index of the second edge.

For test relative to a string.

`(is <isStringType> <string>)`

where:

- `<isStringType>`: The type of query to perform.
- `<string>`: The string to check.

For test relative to a graph element type.

`(is <isGraphType> <siteType>)`

where:

- `<isGraphType>`: The type of query to perform.
- `<siteType>`: The graph element type [Cell (or Vertex if using intersections)].

For test about hidden information.

```
(is <isIndexPlayerType> [<siteType>] <int> (<int> | <roleType>))
```

where:

- `<isIndexPlayerType>`: The type of query to perform.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `<int>`: The site to check.
- `<int>`: The index of the player.
- `<roleType>`: The role of the player.

For test about a single integer.

```
(is <isIntegerType> [<int>])
```

where:

- `<isIntegerType>`: The type of query to perform.
- `[<int>]`: The value.

For tests relative to a component.

```
(is <isComponentType> [<int>] [<siteType>] ([at:<int>] |
     [in:<region>]) [<moves>])
```

where:

- `<isComponentType>`: The type of query to perform.
- `[<int>]`: The piece possibly under threat.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[at:<int>]`: The location of the piece to check.
- `[in:<region>]`: The locations of the piece to check.
- `[<moves>]`: The specific moves used to threat.

For testing the relation between two sites.

```
(is Related <relationType> [<siteType>] <int> (<int> | <region>))
```

where:

- `<relationType>`: The type of relation to check between the graph elements.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `<int>`: The first site.
- `<int>`: The second site.

- <region>: The region of the second site.

For testing a region.

```
(is Target ([<int>] | [<string>]) {int} ([int] | [{int}]))
```

where:

- [<int>]: The index of the container [0].
- [<string>]: The name of the container [”Board”].
- {int}: The configuration defined by the indices of each piece.
- [int]: The specific site of the configuration.
- [{int}]: The specific sites of the configuration.

For test relative to a connection.

```
(is <isConnectType> [<siteType>] [<int>] [at:<int>] ({<region>} |
     <roleType> | <regionTypeStatic>))
```

where:

- <isConnectType>: The type of query to perform.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: The minimum number of set need to connect [number of owned regions if not specified].
- [at:<int>]: The specific starting position need to connect.
- {<region>}: The disjointed regions set, which need to use for connection.
- <roleType>: The role of the player.
- <regionTypeStatic>: Type of the regions.

For test relative to a line.

```
(is Line [<siteType>] <int> [<absoluteDirection>] ([through:<int>]
     | [throughAny:<region>]) ([<roleType>] ([what:<int>]
     | [whats:{<int>}]) [exact:<boolean>] [if:<boolean>]
     [byLevel:<boolean>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- <int>: Minimum length of lines.
- [<absoluteDirection>]: Direction category to which potential lines must belong [Adjacent].

- [through:<int>]: Location through which the line must pass.
- [throughAny:<region>]: The line must pass through at least one of these sites.
- [<roleType>]: The owner of the pieces making a line.
- [what:<int>]: The index of the component composing the line.
- [whats:{<int>}]: The indices of the components composing the line.
- [exact:<boolean>]: If true, then lines cannot exceed minimum length [false].
- [if:<boolean>]: The condition on each site on the line [true].
- [byLevel:<boolean>]: If true, then lines are detected in using the level in a stack [false].

For test relative to a loop.

```
(is Loop [<siteType>] ([surround:<roleType>] | [{<roleType>}])
     [<absoluteDirection>] [<int>] ([<int>] | [<region>])
     [path:<boolean>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [surround:<roleType>]: Used to define the inside condition of the loop.
- [{<roleType>}]: The list of items inside the loop.
- [<absoluteDirection>]: The direction of the connection [Adjacent].
- [<int>]: The colour of the loop.
- [<int>]: The starting point of the loop.
- [<region>]: The region to start to detect the loop.
- [path:<boolean>]: Whether to detect loops in the paths of pieces (e.g. Trax).

For test relative a path.

```
(is Path <siteType> (<player> | <roleType>) ([length:<int>]
     ([maxLimit:<int>]) [closed:<boolean>])
```

where:

- <siteType>: The graph element type [Cell (or Vertex if using intersections)].
- <player>: The current player.
- <roleType>: The role of the player.
- [length:<int>]: The exact component size.
- [maxLimit:<int>]: The maximum component size [1 to range].
- [closed:<boolean>]: Is used to detect closed components.

For test relative to an empty or occupied site.

`(is <isSiteType> [<siteType>] <int>)`

where:

- `<isSiteType>`: The type of query to perform.
- `[<siteType>]`: Graph element type [Cell (or Vertex if using intersections)].
- `<int>`: The index of the site.

For testing if a site is in a region.

`(is In ([<int>] | [{<int>}]) <region>)`

where:

- `[<int>]`: The site [(to)].
- `[{<int>}]`: The sites.
- `<region>`: The region.

Examples

```
(is Tree Mover)

(is SpanningTree Mover)

(is CaterpillarTree Mover)

(is TreeCentre Mover)

(is RegularGraph Mover)

(is Enemy (who at:(last To)))

(is Prev Mover)

(is Triggered "Checkmate" Next)

(is Cycle)

(is Full)

(is Crossing (last To) (to))

(is Decided "End")

(is Proposed "End")

(is LastFrom Vertex)

(is Masked (last To) Mover)

(is Even (last To))

(is Visited (last To))

(is Threatened (id "King" Mover) at:(to))

(is Related Adjacent (from) (sites Occupied by:Next))

(is Target {2 2 2 2 0 0 1 1 1 1})

(is Blocked Mover)

(is Connected Mover)

(is Connected { (sites Side S) (sites Side NW) (sites Side NE) })

(is Line 3)

(is Line 5 Orthogonal if:(not (is In (to) (sites Mover))))

(is Loop)

(is Loop (mover) path:true)

(is Path Edge Mover length:4 closed:false)

(is Empty (to))

(is Occupied Vertex (to))

(is In { (last To) (last From) } (sites Mover))
```

7.6.2 isComponentType

Defines the types of Is test according to a component and a site/region.

Value	Description
Threatened	To check if a location is under threat.
Within	To check if a specific piece is on the designed region.

7.6.3 isConnectType

Defines the types of Is for a connected or blocked test.

Value	Description
Connected	To check if a location is under threat.
Blocked	To check if a specific piece is on the designed region.

7.6.4 isGraphType

Defines the types of Is test according to a graph element.

Value	Description
LastFrom	Check the graph element type of the “from” location of the last move.
LastTo	Check the graph element type of the “to” location of the last move.

7.6.5 isIndexPlayerType

Defines the types of Is test according to an index and a player.

Value	Description
Visible	To check if a specific site is visible to a specific player.
Masked	To check if a specific site is masked to a specific player.
Invisible	To check if a specific site is invisible to a specific player.

7.6.6 isIntegerType

Defines the types of Is test according to an integer.

Value	Description
Odd	To check if a value is odd.
Even	To check if a value is even.
Visited	To check if a site was already visited by a piece in the same turn.
SidesMatch	To detect whether the terminus of a tile matches with its neighbors.
PipsMatch	To detect whether the pips of a domino match its neighbours.

Flat	To Ensures that in a 3D board, all the pieces in the bottom layer must be placed so that they do not fall.
AnyDie	To check if any current die is equal to a specific value.

7.6.7 isPlayerType

Defines the types of Is test for a player.

Value	Description
Mover	To check if a player is the mover.
Next	To check if a player is the next mover.
Prev	To check if a player is the previous mover.
Friend	To check if a player is the friend of the mover.
Enemy	To check if a player is the enemy of the mover.
Active	To check if a player is active.

7.6.8 isSimpleType

Defines the types of Is test for a player with no parameter.

Value	Description
Cycle	To check if the game is repeating the same set of states three times with exactly the same moves during these states.
Pending	To check if the state is in pending.
Full	To check if the board is full.

7.6.9 isSiteType

Defines the types of Is test for a site.

Value	Description
Empty	To check if a site is empty.
Occupied	To check if a site is occupied.

7.6.10 isStringType

Defines the types of Is test according to a String parameter.

Value	Description
Proposed	To check if a specific proposition was made.
Decided	To check if a specific proposition was decided.

7.6.11 isTreeType

Defines the types of Is test for a regular graph.

Value	Description
Tree	To check if the induced graph (by adding or deleting edges) is a tree or not.
SpanningTree	To check if the induced graph (by adding or deleting edges) is a spanning tree or not.
CaterpillarTree	To check if the induced graph (by adding or deleting edges) is the largest caterpillar Tree or not.
TreeCentre	To check whether the last vertex is the centre of the tree (or sub tree).

7.7 Math

Math queries return a boolean result based on given inputs.

7.7.1 and

Returns whether all specified conditions are true.

Format

For an and between two booleans.

`(and <boolean> <boolean>)`

where:

- `<boolean>`: First condition.
- `<boolean>`: Second condition.

For an and between many booleans.

`(and {<boolean>})`

where:

- `{<boolean>}`: The list of conditions to check.

Examples

```
(and (= (who at:(last To)) (mover)) (!= (who at:(last From)) (mover)))

(and
    {
        (= (who at:(last To)) (mover))
        (!= (who at:(last From)) (mover))
        (is Pending)
    }
)
```

Remarks

This test returns false as soon as any of its conditions return false, so it pays to test conditions that are faster and more likely to fail first.

7.7.2 = (equals)

Tests if valueA = valueB, if all the integers in the list are equals, or if the result of the two regions functions are equals.

Format

For testing if two int functions are equals. Also use to test if the index of a roletype is equal to an int function.

(= <int> (<int> | <roleType>))

where:

- <int>: The first value.
- <int>: The second value.
- <roleType>: The second owner value of this role.

For test if two regions are equals.

(= <region> <region>)

where:

- <region>: The first region function.
- <region>: The second region function.

Examples

```
(= (mover) 1)

(= (sites Occupied by:Mover) (sites Next))
```

7.7.3 >= (ge)

Tests if valueA $\geq$ valueB.

Format

(>= <int> <int>)

where:

- <int>: The left value.
- <int>: The right value.

Example

```
(>= (mover) (next))
```

7.7.4 > (gt)

Tests if valueA > valueB.

Format

```
(> <int> <int>)
```

where:

- `<int>`: The left value.
- `<int>`: The right value.

Example

```
(> (mover) (next))
```

7.7.5 if

Tests if the condition is true, the function returns the first value, if not it returns the second value.

Format

```
(if <boolean> <boolean> [<boolean>])
```

where:

- `<boolean>`: The condition to check.
- `<boolean>`: The integer returned if the condition is true.
- `[<boolean>]`: The integer returned if the condition is false.

Example

```
(if (is Mover (next)) (is Pending))
```

7.7.6 <= (le)

Tests if valueA $\leq$ valueB.

Format

```
(<= <int> <int>)
```

where:

- <int>: The left value.
- <int>: The right value.

Example

```
(<= (mover) (next))
```

7.7.7 < (lt)

Tests if valueA $<$ valueB.

Format

```
(< <int> <int>)
```

where:

- <int>: The left value.
- <int>: The right value.

Example

```
(< (mover) (next))
```

7.7.8 not

Tests the not condition.

Format

```
(not <boolean>)
```

where:

- `<boolean>`: The condition.

Example

```
(not (is In (last To) (sites Mover)))
```

7.7.9 ! = (notEqual)

Tests if valueA $\neq$ valueB.

Format

For testing if two int functions are not equals. Also use to test if the index of a roletype is not equal to an int function.

```
(!= <int> (<int> | <roleType>))
```

where:

- `<int>`: The first value.
- `<int>`: The second value.
- `<roleType>`: The second owner value of this role.

For test if two regions are not equals.

```
(!= <region> <region>)
```

where:

- `<region>`: The left value.
- `<region>`: The right value.

Examples

```
(!= (mover) (next))

(!= (sites Occupied by:Mover) (sites Mover))
```

7.7.10 or

Tests the Or boolean node. True if at least one condition is true between the two conditions.

Format

For an or between two booleans.

`(or <boolean> <boolean>)`

where:

- `<boolean>`: First condition.
- `<boolean>`: Second condition.

For an or between many booleans.

`(or {<boolean>})`

where:

- `{<boolean>}`: The list of conditions.

Examples

```
(or (= (who at:(last To)) (mover)) (!= (who at:(last From)) (mover)))

(or
    {
        (= (who at:(last To)) (mover))
        (!= (who at:(last From)) (mover))
        (is Pending)
    }
)
```

7.7.11 xor

Tests the Xor boolean node.

Format

`(xor <boolean> <boolean>)`

where:

- `<boolean>`: First condition.

- <boolean>: Second condition.

Example

```
(xor (= (who at:(last To)) (mover)) (!= (who at:(last From)) (mover)))
```

7.8 No

The `(no ...)` 'super' ludeme returns whether a given query about the game state is false. Such queries might include whether there are no moves available to the current player.

7.8.1 no

Returns whether a certain query about the game state is false.

Format

```
(no Moves <roleType>)
```

where:

- `<roleType>`: The role of the player.

Example

```
(no Moves Mover)
```

7.9 Was

The `(was ...)` 'super' ludeme returns a true/false result as to whether a certain event has occurred in the game, for example whether the last move was a pass.

7.9.1 was

Returns whether a specified event has occurred in the game.

Format

```
(was Pass)
```

Example

```
(was Pass)
```

8
Integer Functions

Integer functions are ludemes that return a single integer value according to some specified function or criteria. They specify the *amount* of certain aspects of the game state. The value returned by the function can be positive or negative.

Care must be taken when dealing with negative values, as they are are typically used to indicate illegal situations within the code. Care must also be taken with large positive return values, as they are uncapped and can be arbitrarily large.

8.1 Board

Board functions return an integer value based on the current board state.

8.1.1 ahead

Returns the site in a given direction from a specified site.

Format

```
(ahead [<siteType>] <int> [steps:<int>] ([<relativeDirection>] |
     [<absoluteDirection>]))
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `<int>`: Source site.
- `[steps:<int>]`: Distance to travel [1].
- `[<relativeDirection>]`: Direction relative to player [Forward].
- `[<absoluteDirection>]`: Absolute compass direction.

Examples

```
(ahead (centrePoint) E)

(ahead (last To) steps:3 N)
```

Remarks

If there is no site in the specified direction, then the index of the source site is returned.

8.1.2 centrePoint

Returns the index of the central board site.

Format

```
(centrePoint [<siteType>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

Example

```
(centrePoint)
```

8.1.3 column

Returns the column number in which a given site lies.

Format

```
(column [<siteType>] of:<int>)
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- of:<int>: The site to check.

Example

```
(column of:(to))
```

Remarks

Returns OFF (-1) if the site does not belong to any column.

8.1.4 coord

Returns the site index of a given board coordinate.

Format

```
(coord [<siteType>] <string>)
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- <string>: The coordinates of the site.

Example

```
(coord "A1")
```

8.1.5 cost

Returns the cost of graph element(s).

Format

```
(cost [<siteType>] (at:<int> | in:<region>))
```

where:
- `[<siteType>]`: The type of the graph element [Cell].
- `at:<int>`: The index of the graph element.
- `in:<region>`: The region of the graph elements.

Example

```
(cost at:(to))
```

8.1.6 handSite

Returns one site of one hand.

Format

```
(handSite (<int> | <roleType>) [<int>])
```

where:
- `<int>`: The index of the owner of the hand.
- `<roleType>`: The roleType of the owner of the hand.
- `[<int>]`: The site on the hand.

Example

```
(handSite Mover)
```

Remarks

To check a specific site of a specific hand.

8.1.7 id

Returns the index of a component, player or region.

Format

To get the index of a component containing the name and owns by who.

```
(id [<string>] [<roleType>])
```

where:

- [<string>]: The name of the component.
- [<roleType>]: The owner of the component.

To get the index of a component containing its name.

```
(id <string>)
```

where:

- <string>: The name of the component.

Examples

```
(id "Pawn" Mover)

(id "Pawn1")
```

Remarks

To translate a component, a player or a region to an index.

8.1.8 layer

Returns the layer of a site.

Format

```
(layer of:<int> [<siteType>])
```

where:

- of:<int>: The site to check.
- [<siteType>]: The graph element type of the site.

Example

```
(layer of:(to))
```

Remarks

This ludeme returns the layer of a site for 3D boards. If the board is flat (2D), then 0 is returned to indicate the board layer.

8.1.9 mapEntry

Returns the value corresponding to a specified entry in a map.

Format

```
(mapEntry [<string>] (<int> | <roleType>))
```

where:

- [<string>]: The name of the map.
- <int>: The key value to check.
- <roleType>: The roleType corresponding to an integer value to check.

Examples

```
(mapEntry (last To))

(mapEntry (trackSite Move steps:(count Pips)))
```

Remarks

Maps are used to stored mappings from one set of numbers to another.

8.1.10 phase

Returns the phase of a graph element on the board.

Format

```
(phase [<siteType>] of:<int>)
```

where:

- `[<siteType>]`: Type of graph element.
- `of:<int>`: The index of the element.

Example

```
(phase of:(last To))
```

Remarks

If the graph element is not on the main board, the ludeme returns (Undefined) -1.

8.1.11 row

Returns the row of a site.

Format

```
(row [<siteType>] of:<int>)
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `of:<int>`: The site to check.

Example

```
(row of:(to))
```

8.1.12 where

Returns the site of a piece if it is on the board, else OFF (-1).

Format

If a piece is on the board, return its site else Off.

```
(where <string> (<int> | <roleType>) [state:<int>] [<siteType>])
```

where:

- `<string>`: The name of the piece (without the number at the end).
- `<int>`: The index of the owner.
- `<roleType>`: The roleType of the owner.
- `[state:<int>]`: The local state of the piece.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

If a piece is on the board, return its site else -1.

```
(where <int> [<siteType>])
```

where:

- `<int>`: The index of the piece.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

Examples

```
(where "Pawn" Mover)

(where (what at:(last To)))
```

Remarks

The name of the piece can be specific without the number on it because the owner is also specified in the ludeme.

8.2 Card

Card functions return an integer value based on the current state of specified `Card` components.

8.2.1 card

Returns a site related to the last move.

Format

For the trump suit of a card.

```
(card TrumpSuit)
```

For the rank, the suit, trump rank or the trump value of a card.

```
(card <cardSiteType> at:<int> [level:<int>])
```

where:

- `<cardSiteType>`: The property to return.
- `at:<int>`: The site where the card is.
- `[level:<int>]`: The level where the card is.

Examples

```
(card TrumpSuit)

(card TrumpValue at:(from) level:(level))

(card TrumpRank at:(from) level:(level))

(card Rank at:(from) level:(level))

(card Suit at:(from) level:(level))
```

8.2.2 cardSiteType

Defines the types of properties which can be returned for the Card super ludeme according an index and optionally a level.

Value	Description
Rank	To return the rank of a card.
Suit	To return the suit of a card.

TrumpValue	To return the value of the trump of a card.
TrumpRank	To return the rank of the trump of a card.

8.3 Connection

Connection functions return an integer value based on connectivity between components in the current game state.

8.3.1 groupProduct

Returns the product of all group sizes of a player.

Format

```
(groupProduct [<siteType>] (<roleType> | <player>))
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `<roleType>`: The roleType of the player owning the components in the group.
- `<player>`: The index of the player.

Example

```
(groupProduct Mover)
```

Remarks

This ludeme is used in games such as Omega.

8.4 Context

Context functions return an integer value based on certain properties of the game's current `Context` object. Context values are typically used as temporary variables during move planning for chaining nontrivial move sequences.

8.4.1 between

Returns the "between" value of the context.

Format

```
(between)
```

Example

```
(between)
```

Remarks

This ludeme identifies the location(s) between the current position of a component and its destination location of a move. It can also represent each site (iteratively) surrounded by other sites or inside a loop. This ludeme is typically used to test a condition or apply an effect to each site "between" other specified sites.

8.4.2 edge

Returns the corresponding edge if both vertices are specified, else returns the current "edge" value from the context.

Format

For returning the index of an edge using the two indices of vertices.

```
(edge <int> <int>)
```

where:

- `<int>`: The first vertex of the edge.
- `<int>`: The second vertex of the edge.

For returning the edge value of the context.

```
(edge)
```

Examples

```
(edge (from) (to))

(edge)
```

Remarks

This ludeme identifies the value of a move applied to an edge.

8.4.3 from

Returns the “from” value of the context.

Format

```
(from [at:<whenType>])
```

where:

- [at:<whenType>]: To return the “from” location at a specific time within the game.

Example

```
(from)
```

Remarks

This ludeme identifies the current position of a specified component. It is used for the component's move generator and for all the decision moves.

8.4.4 hint

Returns the “hint” value of the context.

Format

```
(hint [<siteType>] [at:<int>])
```

where:

- [<siteType>]: The type of the site to look.
- [at:<int>]: The index of the site.

Example

```
(hint)
```

Remarks

This ludeme identifies the hint position of a deduction puzzle stored in the context.

8.4.5 level

Returns the “level” value of the context.

Format

```
(level)
```

Example

```
(level)
```

Remarks

This ludeme identifies the level of the current position of a component on a site that is stored in the context. It is used for stacking games and to generate the moves of the components and for all decision moves.

8.4.6 site

Returns the “site” value stored in the context.

Format

```
(site)
```

Example

```
(site)
```

Remarks

This ludeme is used by `(forEach Site ...)` to iterate over a set of sites.

8.4.7 to

Returns the "to" value of the context.

Format

```
(to)
```

Example

```
(to)
```

Remarks

This ludeme returns the destination location the current component is moving to. It is used to generate component moves and for all decision moves.

8.4.8 track

Returns the "track" value of the context.

Format

```
(track)
```

Example

```
(track)
```

Remarks

Used in a (forEach Track ...) ludeme to set the value to the index of each track.

8.4.9 var

Returns the value stored in the var variable from the context.

Format

```
(var [<string>])
```

where:

- `[<string>]`: The key String value to check.

Example

```
(var "current")
```

Remarks

To identify the value stored previously with a key in the context. If no key specified, the var variable of the context is returned.

8.5 Count

Count is a 'super' ludeme that returns the count of a specified property within the game, such as the number of players, components, sites, turns, groups, etc.

8.5.1 count

Returns the count of the specified property.

Format

For counting according to no parameters or only a graph element type.

```
(count <countSimpleType> [<siteType>])
```

where:

- `<countSimpleType>`: The property to count.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

For counting according to a site or a region.

```
(count [<countSiteType>] [<siteType>] ([in:<region>] | [at:<int>] |
    [<string>]))
```

where:

- `[<countSiteType>]`: The property to count.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[in:<region>]`: The region to count.
- `[at:<int>]`: The site from which to compute the count [(last To)].
- `[<string>]`: The name of the container from which to count the number of sites or the name of the piece to count only pieces of that type.

For counting according to a component.

```
(count <countComponentType> [<siteType>] ([<roleType>] | [of:<int>])
    [<string>] [in:<region>])
```

where:

- `<countComponentType>`: The property to count.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[<roleType>]`: The role of the player [All].
- `[of:<int>]`: The index of the player.

- [<string>]: The name of the container from which to count the number of sites or the name of the piece to count only pieces of that type.
- [in:<region>]: The region where to count the pieces.

For counting elements in a group.

```
(count Groups [<siteType>] ([<roleType>] | [of:<int>]) [min:<int>]
     [<absoluteDirection>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<roleType>]: The role of the player [All].
- [of:<int>]: The index of the player.
- [min:<int>]: Minimum size of each group [0].
- [<absoluteDirection>]: Direction of connection [Adjacent].

For counting elements in a region of liberties.

```
(count Liberties [<siteType>] ([in:<region>] | [at:<int>])
     [<absoluteDirection>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [in:<region>]: The region from which to compute the count.
- [at:<int>]: The site from which to compute the count [(last To)].
- [<absoluteDirection>]: Direction of connection [Adjacent].

For counting the number of steps between two sites.

```
(count Steps [<siteType>] [<relationType>] [<step>] <int> <int>)
```

where:

- [<siteType>]: Graph element type [Cell (or Vertex if using intersections)].
- [<relationType>]: The relation type of the steps [Adjacent].
- [<step>]: Define a particular step move to step.
- <int>: The first site.
- <int>: The second site.

Examples

```
(count Players)

(count Vertices)

(count Moves)

(count at:(last To))

(count Sites in:(sites Empty))

(count Pieces Mover)

(count Pips)

(count Groups Orthogonal)

(count Liberties Orthogonal)

(count Steps (where (id "King")) (where (id "Queen")))
```

8.5.2 countComponentType

Defines the types of components that can be counted within a game.

Value	Description
Pieces	Number of pieces on the board (or in hand), per player or over all players.
Pips	The number of pips showing on all dice, or dice owned by a specified player.

8.5.3 countSimpleType

Defines the types of properties that can be counted without a parameter (apart from the graph element type, where relevant).

Value	Description
Rows	Number of rows on the board.
Columns	Number of columns on the board.
Turns	Number of turns played so far in this trial.
Moves	Number of moves made so far in this trial.
Trials	Number of completed games within a match.
MovesThisTurn	Number of moves made so far this turn.
Phases	Number of phase changes during this trial.
Vertices	Number of adjacent (connected) elements.

Edges	Number of edges on the board.
Cells	Number of cells on the board.
Players	Number of players.
Active	Number of active players.

8.5.4 countSiteType

Defines the types of sites that can be counted within a game.

Value	**Description**
Sites	Number of playable sites within a region or container.
Adjacent	Number of adjacent (connected) elements.
Neighbours	Number of neighbours (not necessarily connected).
Orthogonal	Number of orthogonal elements.
Diagonal	Number of diagonal elements.
Off	Number of off-diagonal elements.

8.6 Dice

Dice functions return an integer value based on the current dice roll.

8.6.1 face

Returns the face of the die according to the current state of the position of the die.

Format

```
(face <int>)
```

where:

- `<int>`: The location of the die.

Example

```
(face (handSite P1))
```

Remarks

To know the face value of a die.

8.6.2 pips

Returns the number of pips of a die.

Format

```
(pips)
```

Example

```
(pips)
```

8.7 Last

Last is a ‘super’ ludeme that returns a site related to the last move.

8.7.1 last

Returns a site related to the last move.

Format

```
(last <lastType> [afterConsequence:<boolean>])
```

where:

- `<lastType>`: The site to return.
- `[afterConsequence:<boolean>]`: To check the from location of the last move after applying the consequence [false].

Examples

```
(last To)

(last From)
```

8.7.2 lastType

Defines the types of Last integer ludeme.

Value	Description
To	To return the “to” site of the last move.
From	To return the “from” site of the last move.

8.8 Match

Match functions return an integer value based on the state of the current match.

8.8.1 matchScore

Returns the match score of a player.

Format

```
(matchScore <roleType>)
```

where:

- `<roleType>`: The roleType of the player.

Example

```
(matchScore P1)
```

Remarks

This is used to know the score of the player in a match.

8.9 Math

Math functions return an integer value based on given inputs.

8.9.1 abs

Return the absolute value of a value.

Format

```
(abs <int>)
```

where:

- `<int>`: The value.

Example

```
(abs (value Piece of:(what at:(to))))
```

8.9.2 + (add)

Adds many values.

Format

To add two values.

```
(+ <int> <int>)
```

where:

- `<int>`: The first value.
- `<int>`: The second value.

To add all the values of a list.

```
(+ {<int>})
```

where:

- `{<int>}`: The list of the values.

Examples

```
(+ (value Piece of:(what at:(from))) (value Piece of:(what at:(to))))

(+
    {
        (value Piece of:(what at:(from)))
        (value Piece of:(what at:(to)))
        (value Piece of:(what at:(between)))
    }
)
```

Remarks

This is used to add many values.

8.9.3 / (div)

To divide a value by another.

Format

```
(/ <int> <int>)
```

where:

- `<int>`: The value to divide.
- `<int>`: To divide by b.

Example

```
(/ (value Piece of:(what at:(from))) (value Piece of:(what at:(to))))
```

Remarks

The result will be an integer.

8.9.4 if

Returns a value according to a condition.

Format

```
(if <boolean> <int> <int>)
```

where:

- `<boolean>`: The condition.
- `<int>`: The value returned if the condition is true.
- `<int>`: The value returned if the condition is false.

Example

```
(if (is Mover P1) 1 2)
```

Remarks

This ludeme is used to get a different int depending on a condition in a value of a ludeme.

8.9.5 max

Returns the maximum of two specified values.

Format

```
(max <int> <int>)
```

where:

- `<int>`: The first value.
- `<int>`: The second value.

Example

```
(max (mover) (next))
```

8.9.6 min

Returns the minimum of two specified values.

Format

```
(min <int> <int>)
```

where:

- `<int>`: The first value.
- `<int>`: The second value.

Example

```
(min (mover) (next))
```

8.9.7 % (mod)

Returns the modulo of a value.

Format

```
(% <int> <int>)
```

where:

- `<int>`: The value.
- `<int>`: The modulo.

Example

```
(% (count Moves) 3)
```

8.9.8 * (mul)

Returns to multiple of values.

Format

For the product of two values.

```
(* <int> <int>)
```

where:

- `<int>`: The first value.

- <int>: The second value.

For the product of many values.

```
(* {<int>})
```

where:

- {<int>}: The list.

Examples

```
(* (mover) (next))

(* { (mover) (next) (prev) })
```

8.9.9 ∧ (pow)

Computes the first parameter to the power of the second parameter.

Format

```
(∧ <int> <int>)
```

where:

- <int>: The value.
- <int>: The power.

Example

```
(^ (value Piece of:(what at:(last To))) 2)
```

8.9.10 – (sub)

Returns the subtraction A minus B.

Format

```
(– [<int>] <int>)
```

where:

- [<int>]: The first value (to subtract from) [0].
- <int>: The second value (to be subtracted from the first value).

Examples

```
(- 1)

(-
    (value Piece of:(what at:(last To)))
    (value Piece of:(what at:(last From)))
)
```

8.10 Size

Size is a 'super' ludeme that returns the size of a specified property within the game, such as a stack, a group or a territory.

8.10.1 size

Returns the size of the specified property.

Format

For the size of a stack.

```
(size Stack [<siteType>] ([in:<region>] | [at:<int>] | [<string>]))
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [in:<region>]: The region to count.
- [at:<int>]: The site from which to compute the count [(last To)].
- [<string>]: The name of the container from which to count the number of sites or the name of the piece to count only pieces of that type.

For the size of a group.

```
(size [Group] [<siteType>] [from:<int>] ([<player>] | [<roleType>])
     [<absoluteDirection>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [from:<int>]: The site to compute the group [(last To)].
- [<player>]: The index of the player owning the components in the group.
- [<roleType>]: The roleType of the player owning the components in the group.
- [<absoluteDirection>]: The type of directions from the site to compute the group [Adjacent].

For the size of a territory.

```
(size [Territory] [<siteType>] (<roleType> | <player>)
     [<absoluteDirection>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].

- `<roleType>`: The roleType of the player owning the components in the territory.
- `<player>`: The index of the player owning the components in the territory.
- [`<absoluteDirection>`]: The type of directions from the site to compute the group [Adjacent].

Examples

```
(size Stack at:(last To))

(size Group from:(last To) Orthogonal)

(size Territory P1)
```

8.11 Stacking

Stacking functions return an integer value based on the state of a specified stack of components.

8.11.1 topLevel

Returns the top level of a stack.

Format

```
(topLevel [<siteType>] at:<int>)
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The site of the stack.

Example

```
(topLevel at:(last To))
```

Remarks

If the game is not a stacking game, then level 0 is returned.

8.12 State

State functions return an integer value based on the current game state.

8.12.1 amount

Returns the amount of a player.

Format

```
(amount (<roleType> | <player>))
```

where:

- `<roleType>`: The role of the player.
- `<player>`: The index of the player.

Example

```
(amount Mover)
```

8.12.2 counter

Returns the automatic counter of the game.

Format

```
(counter)
```

Example

```
(counter)
```

Remarks

To use a counter automatically incremented at each move done, this can be set to another value by the move (setCounter).

8.12.3 mover

Returns the index of the current player.

Format

```
(mover)
```

Example

```
(mover)
```

Remarks

To apply some specific condition/rules to the current player.

8.12.4 next

Returns the index of the next player.

Format

```
(next)
```

Example

```
(next)
```

Remarks

This ludeme is used to apply some specific condition or rule to the next player.

8.12.5 pot

Returns the pot of the game.

Format

```
(pot)
```

Example

```
(pot)
```

8.12.6 prev

Returns the index of the previous player.

Format

```
(prev)
```

Example

```
(prev)
```

Remarks

To apply some specific conditions/rules to the previous player.

8.12.7 score

Returns the score of one specific player.

Format

```
(score (<player> | <roleType>))
```

where:

- <player>: The index of the player.
- <roleType>: The roleType of the player.

Example

```
(score Mover)
```

8.12.8 state

Returns the local state value of a specified site.

Format

```
(state [<siteType>] at:<int> [level:<int>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The location to check.
- `[level:<int>]`: The level to check [0].

Example

```
(state at:(last To))
```

Remarks

Thisludeme is used for games with local state values associated with sites.

8.12.9 what

Returns the index of the component at a specific location/level.

Format

```
(what [<siteType>] at:<int> [level:<int>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The location to check.
- `[level:<int>]`: The level to check [0].

Example

```
(what at:(last To))
```

8.12.10 who

Returns the index of the owner at a specific location/level.

Format

```
(who [<siteType>] at:<int> [level:<int>])
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `at:<int>`: The location to check.
- `[level:<int>]`: The level to check.

Example

```
(who at:(last To))
```

8.13 Tile

Tile functions return an integer value based on the current state of specified `Tile` components.

8.13.1 pathExtent

Returns the maximum extent of a path.

Format

```
(pathExtent [<int>] ([<int>] | [<region>]))
```

where:

- `[<int>]`: The colour of the path [(mover)].
- `[<int>]`: The starting point of the path [(lastTo)].
- `[<region>]`: The starting points of the path [(regionLastToMove)].

Example

```
(pathExtent (mover))
```

Remarks

The path extent is the maximum board width and/or height that the path extends to. This is used in tile-based games with paths, such as Trax.

8.14 TrackSite

TrackSite is a 'super' ludeme that returns a site on a track.

8.14.1 trackSite

Returns a site on a track.

Format

For the last site in a track.

```
(trackSite EndSite ([<player>] | [<roleType>]) [<string>])
```

where:

- [<player>]: The index of the player.
- [<roleType>]: The role of the player.
- [<string>]: The name of the track ["Track"].

For getting the site in a track from a site after some steps.

```
(trackSite Move [from:<int>] ([<roleType>] | [<player>] | [<string>])
     steps:<int>)
```

where:

- [from:<int>]: The current location [(from)].
- [<roleType>]: The role of the owner of the track [Mover].
- [<player>]: The owner of the track [(mover)].
- [<string>]: The name of the track.
- steps:<int>: The distance to move on the track.

Examples

```
(trackSite EndSite)

(trackSite Move steps:(count Pips))
```

8.15 Value

Value is a 'super' ludeme that returns the value of a specified property within the game, such as a player or a component.

8.15.1 value

Returns the value of the specified property.

Format

For returning the pending value.

```
(value Pending)
```

For returning the player value.

```
(value Player (<int> | <roleType>))
```

where:

- `<int>`: The index of the player.
- `<roleType>`: The roleType of the player.

For returning the piece value.

```
(value Piece of:<int>)
```

where:

- `of:<int>`: The index of the component.

Examples

```
(value Pending)

(value Player (who at:(to)))

(value Piece of:(what at:(to)))
```

9

Integer Array Functions

Integer array functions are ludemes that return an array of integer values, typically for processing by other ludemes.

9.1 Math

Math int array functions return an of integer values based on given inputs.

9.1.1 difference

Returns the difference between two arrays of integers, i.e. the elements in A that are not in B.

Format

```
(difference {<int>} ({<int>} | <int>))
```

where:

- {<int>}: The original array.
- {<int>}: The array to remove from the original array.
- <int>: The integer to remove from the original array.

Example

```
(difference {1 2 3 4} { 2 3 })
```

9.2 State

State int array functions return integer values based on the current game state.

9.2.1 rotations

Returns the list of rotation indices according to a tiling type.

Format

```
(rotations (<absoluteDirection> | {<absoluteDirection>}))
```

where:

- <absoluteDirection>: The direction of the possible rotations.
- {<absoluteDirection>}: The directions of the possible rotations.

Example

```
(rotations Orthogonal)
```

10
Region Functions

Region functions are ludemes that return *regions* composed of collections of sites. These can be *static* regions defined in the game's `equipment`, such as player homes or special target regions, or *dynamic* regions calculated on-the-fly during play according to the current game state, such as the region of unoccupied sites or the region of sites occupied by particular player or piece type.

10.1 Filter

Filter region functions return a region of sites based on a specified iterator that is applied to all board sites.

10.1.1 forEach

Returns the sites satisfying a constraint from a given region.

Format

```
(forEach <region> if:<boolean>)
```

where:

- `<region>`: The original region.
- `if:<boolean>`: The condition to satisfy.

Example

```
(forEach (sites Occupied by:P1) if:(= (what at:(to)) (id "Pawn1")))
```

10.2 Math

Math region functions return a combined region of sites based on provided input regions.

10.2.1 difference

Returns the set difference, i.e. elements of the source region are not in the subtraction region.

Format

```
(difference <region> (<region> | <int>))
```

where:

- `<region>`: The original region.
- `<region>`: The region to remove from the original.
- `<int>`: The site to remove from the original.

Example

```
(difference (sites Occupied by:Mover) (sites Mover))
```

10.2.2 expand

Expands a given region/site in all directions the specified number of steps.

Format

```
(expand ([<int>] | [<string>]) (<region> (origin:<int>) [steps:<int>]
     [<absoluteDirection>] [<siteType>])
```

where:

- `[<int>]`: The index of the container.
- `[<string>]`: The name of the container.
- `<region>`: The region.
- `origin:<int>`: The site.
- `[steps:<int>]`: The distance to expand [steps:1].
- `[<absoluteDirection>]`: The absolute direction to expand.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

Example

```
(expand (sites Bottom) steps:2)
```

10.2.3 if

Returns a region when the condition is satisfied and another when it is not.

Format

```
(if <boolean> <region> [<region>])
```

where:

- <boolean>: The condition to satisfy.
- <region>: The region returned when the condition is satisfied.
- [<region>]: The region returned when the condition is not satisfied.

Example

```
(if (is Mover P1) (sites P1) (sites P2))
```

10.2.4 intersection

Returns the intersection of many regions.

Format

For the intersection of two regions.

```
(intersection <region> <region>)
```

where:

- <region>: The first region.
- <region>: The second region.

For the intersection of many regions.

```
(intersection {<region>})
```

where:

- {<region>}: The different regions.

Examples

```
(intersection (sites Mover) (sites Occupied by:Mover))

(intersection
    { (sites Mover) (sites Occupied by:Mover) (sites Occupied by:Next) }
)
```

10.2.5 union

Merges many regions into one.

Format

For the union of two regions.

`(union <region> <region>)`

where:

- <region>: The first region.
- <region>: The second region.

For the union of many regions.

`(union {<region>})`

where:

- {<region>}: The different regions.

Examples

```
(union (sites P1) (sites P2))

(union { (sites P1) (sites P2) (sites P3) })
```

10.3 Sites

The (sites ...) 'super' ludeme returns a set of sites of the specified type, such as board sites, hand sites, corners, edges, empty sites, playable sites, etc.

10.3.1 sites

Returns the specified set of sites.

Format

For getting a random site in a region.

```
(sites Random [<region>] [num:<int>])
```

where:

- [<region>]: The region to get [(sites Empty Cell)].
- [num:<int>]: The number of sites to return [1].

For getting the sites crossing another site.

```
(sites Crossing at:<int> ([<player>] | [<roleType>]))
```

where:

- at:<int>: The specific starting position needs to crossing check.
- [<player>]: The returned crossing items player type.
- [<roleType>]: The returned crossing items player role type.

For getting the site of a group.

```
(sites Group [<siteType>] at:<int> [<direction>] [if:<boolean>])
```

where:

- [<siteType>]: The type of the graph elements of the group.
- at:<int>: The specific starting position needs to connect.
- [<direction>]: The directions of the connection between elements in the group [Adjacent].
- [if:<boolean>]: The condition on the pieces to include in the group.

For the sites relative to edges.

```
(sites <sitesEdgeType>)
```

where:

- `<sitesEdgeType>`: Type of sites to return.

For getting sites without any parameter or only the graph element type.

```
(sites <sitesSimpleType> [<siteType>])
```

where:

- `<sitesSimpleType>`: Type of sites to return.
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

For getting sites according to their coordinates.

```
(sites [<siteType>] {<string>})
```

where:

- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `{<string>}`: The sites corresponding to these coordinates.

For getting sites based on the "from" or "to" locations of all the moves in a collection of moves.

```
(sites <sitesMoveType> <moves>)
```

where:

- `<sitesMoveType>`: Type of sites to return.
- `<moves>`: The moves for which to collect the positions.

For creating a region with specific sites.

```
(sites {<int>})
```

where:

- `{<int>}`: The sites of the region.

For getting sites of a walk.

```
(sites [<siteType>] [<int>] {{<stepType>}} [rotations:<boolean>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: The location from which to compute the walk [(from)].
- {{<stepType>}}: The different turtle steps defining a graphic turtle walk.
- [rotations:<boolean>]: True if the move includes all the rotations of the walk [true].

For getting sites belonging to a part of the board.

```
(sites <sitesIndexType> [<siteType>] [<int>])
```

where:

- <sitesIndexType>: Type of sites to return.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<int>]: Index of the row, column or phase to return. This can also be the value of the local state for the State SitesIndexType or the container to search for the Empty SitesIndexType.

For getting sites of a side of the board.

```
(sites Side [<siteType>] ([<player>] | [<roleType>] |
     [<compassDirection>]))
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<player>]: Index of the player or the component.
- [<roleType>]: The Role type corresponding to the index.
- [<compassDirection>]: Direction of the side to return.

For getting sites at a specific distance of another.

```
(sites Distance [<siteType>] [<relationType>] from:<int> <int>)
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<relationType>]: The relation type of the steps [Adjacent].
- from:<int>: Index of the site.
- <int>: Distance from the site.

For getting sites of a region defined in the equipment or of a single coordinate.

```
(sites ([<player>] | [<roleType>]) [<siteType>] [<string>])
```

where:

- [<player>]: Index of the player or the component.
- [<roleType>]: The Role type corresponding to the index.
- [<siteType>]: The graph element type of the coordinate is specified.
- [<string>]: The name of the region to return or of a single coordinate.

For getting sites relative to a player.

```
(sites <sitesPlayerType> [<siteType>] ([<player>] | [<roleType>])
     [<nonDecision>] [<string>])
```

where:

- <sitesPlayerType>: Type of sites to return.
- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- [<player>]: Index of the player or the component.
- [<roleType>]: The Role type corresponding to the index.
- [<nonDecision>]: Rules used to generate moves for finding winning sites.
- [<string>]: The name of the board or region to return.

For getting sites relative to a player.

```
(sites Start <piece>)
```

where:

- <piece>: Index of the player or the component.

For getting sites occupied by player(s).

```
(sites Occupied (by:<player> | by:<roleType>) ([container:<int>]
     ([container:<string>]) ([component:<int>] ([component:<string>] |
     [components:{<string>}]) [top:<boolean>] [<siteType>])
```

where:

- by:<player>: The roleType of the owner.
- by:<roleType>: The roleType of the owner.
- [container:<int>]: The name of the container.
- [container:<string>]: The name of the container.

- [component:<int>]: The name of the component.
- [component:<string>]: The name of the component.
- [components:{<string>}]: The names of the component.
- [top:<boolean>]: True to look only the top of the stack [true].
- [<siteType>]: The type of the graph element [Vertex].

For getting sites incident to another.

```
(sites Incident <siteType> of:<siteType> at:<int> ([owner:<player>] |
     [<roleType>]))
```

where:

- <siteType>: The graph type of the result.
- of:<siteType>: The graph type of the index.
- at:<int>: Index of the element to check.
- [owner:<player>]: The owner of the site to return.
- [<roleType>]: The role of the owner of the site to return.

For getting sites around another.

```
(sites Around [<siteType>] (<int> | <region>) [<regionTypeDynamic>]
     [distance:<int>] [<absoluteDirection>] [if:<boolean>]
     [includeSelf:<boolean>])
```

where:

- [<siteType>]: The graph element type [Cell (or Vertex if using intersections)].
- <int>: The location to check.
- <region>: The region to check.
- [<regionTypeDynamic>]: The type of the dynamic region.
- [distance:<int>]: The distance around which to check [1].
- [<absoluteDirection>]: The directions to use [Adjacent].
- [if:<boolean>]: The condition to satisfy around the site to be included in the result.
- [includeSelf:<boolean>]: True if the origin site/region is included in the result [false].

For getting sites in a direction from another.

```
(sites Direction (from:<int> | from:<region>) [<direction>]
     [included:<boolean>] [stop:<boolean>] [distance:<int>]
     [<siteType>])
```

where:

- `from:<int>`: The origin region location.
- `from:<region>`: The origin region location.
- `[<direction>]`: The directions of the move [Adjacent].
- `[included:<boolean>]`: True if the origin is included in the result [false].
- `[stop:<boolean>]`: When the condition is true in one specific direction, sites are no longer added to the result [false].
- `[distance:<int>]`: The distance around which to check [Infinite].
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].

For getting sites in the line of sight.

```
(sites LineOfSight [<lineOfSightType>] [<siteType>] [at:<int>]
     [<direction>])
```

where:

- `[<lineOfSightType>]`: The line-of-sight test to apply [Piece].
- `[<siteType>]`: The graph element type [Cell (or Vertex if using intersections)].
- `[at:<int>]`: The location [(last To)].
- `[<direction>]`: The directions of the move [Adjacent].

Examples

```
(sites Random)
(sites Crossing at:(last To) All)
(sites Group Vertex at:(site))
(sites Axial)
(sites Top)
(sites Playable)
(sites Right Vertex)
(sites {"A1" "B1" "A2" "B2"})
(sites From (forEach Piece))
(sites To (forEach Piece))
(sites {1..10})
(sites {1 5 10})
(sites { { F F R F } { F F L F } })
(sites Row 1)
(sites Side NE)
(sites Distance from:(last To) 5)
(sites P1)
(sites "E5")
(sites Hand Mover)
(sites Winning Next (add (to (sites Empty))))
(sites Start (piece (what at:(from))))
(sites Occupied by:Mover)
(sites Incident Edge of:Vertex at:(last To))
(sites Incident Cell of:Edge at:(last To) Mover)
(sites Around (last To))
(sites Around (to) Orthogonal if:(is Empty (to)))
(sites Direction from:(last To) Diagonal)
(sites LineOfSight Orthogonal)
```

10.3.2 sitesEdgeType

Specifies set of edge sites.

Value	Description
Axial	The axials edges sites.
Horizontal	The horizontal edges sites.
Vertical	The vertical edges sites.
Angled	The angled edges sites.
Slash	The slash edges sites.
Slosh	The slosh edges sites.

10.3.3 sitesIndexType

Specifies sets of board sites by some indexed property.

Value	Description
Row	Sites in a specified row.
Column	Sites in a specified column.
Phase	Sites in a specified phase.
Cell	Vertices that make up a cell.
Edge	End points of an edge.
State	Sites with a specified state value.
Empty	Empty (i.e. unoccupied) sites of a container.

10.3.4 sitesMoveType

Specifies sets of sites based on the positions of moves.

Value	Description
From	From-positions of a collection of moves as a set of sites.
To	To-positions of a collection of moves as a set of sites.

10.3.5 sitesPlayerType

Specifies sets of sites associated with given players.

Value	Description
Hand	Sites in a player's hand.
Track	Sites in a player's track.
Winning	Sites that would be winning moves for the current player.
Visible	Sites that are visible for a player.

Masked	Sites that are masked for a player.
Invisible	Sites that are invisible for a player.

10.3.6 sitesSimpleType

Specifies set of sites that do not require any parameters (apart from the graph element type).

Value	**Description**
Board	All board sites.
Top	Sites on the top side of the board.
Bottom	Sites on the bottom side of the board.
Left	Sites on the left side of the board.
Right	Sites on the right side of the board.
Inner	Interior board sites.
Outer	Outer board sites.
Corners	Corner board sites.
ConcaveCorners	Concave corner board sites.
ConvexCorners	Convex corner board sites.
Major	Major generator board sites.
Minor	Minor generator board sites.
Centre	Centre board site(s).
Hint	Sites that contain a puzzle hint.
ToClear	Sites to remove at the end of a capture sequence.
LineOfPlay	Sites in the line of play. Applies to domino game (returns an empty region for other games).
Pending	Sites with a non-zero “pending” value in the game state.
Playable	Playable sites of a boardless game. For other games, returns the set of empty sites adjacent to occupied sites.
LastTo	The set of “to” sites of the last move.
LastFrom	The set of “from” sites of the last move.

10.4 Sites - LineOfSight

The `(sites LineOfSight...)` option of the `(sites ...)` “super” ludeme returns a set of sites based on line-of-site from some specified site.

10.4.1 lineOfSightType

Specifies the expected types of line of sight tests.

Value	Description
Empty	Empty sites in line of sight along each direction.
Farthest	Farthest empty site in line of sight along each direction.
Piece	First piece (of any type) in line of sight along each direction.

11
Direction Functions

Direction functions are ludemes that return an array of integers representing known direction types, according to some specified criteria. All types listed in this chapter may be used for `<directionsFunction>` parameters in other ludemes.

11.1 Difference

The base `directions` function converts input directions to player-centric equivalents.

11.1.1 difference

Returns the difference of two set of directions.

Format

```
(difference <direction> <direction>)
```

where:

- `<direction>`: The original directions.
- `<direction>`: The directions to remove.

Example

```
(difference Orthogonal N)
```

11.2 Directions

The base `directions` function converts input directions to player-centric equivalents.

11.2.1 directions

Converts the directions with absolute directions or relative directions according to the direction of the piece/player to a list of integers.

Format

For defining directions with absolute directions.

```
(directions (<absoluteDirection> | {<absoluteDirection>}))
```

where:

- `<absoluteDirection>`: The absolute direction.
- `{<absoluteDirection>}`: The absolute directions.

For defining directions with relative directions.

```
(directions ([<relativeDirection>] | [{<relativeDirection>}])
     [of:<relationType>] [bySite:<boolean>])
```

where:

- `[<relativeDirection>]`: The relative direction.
- `[{<relativeDirection>}]`: The relative directions.
- `[of:<relationType>]`: The type of directions to return [Adjacent].
- `[bySite:<boolean>]`: If true, the directions to return are computed according to the supported directions of the site and if not according to all the directions supported by the board [False].

Examples

```
(directions Orthogonal)

(directions Forwards)
```

11.3 If

The base `directions` function converts input directions to player-centric equivalents.

11.3.1 if

Returns whether the specified directions are satisfied for the current game.

Format

```
(if <boolean> <direction> <direction>)
```

where:

- `<boolean>`: The condition to verify.
- `<direction>`: The directions if the condition is verified.
- `<direction>`: The directions if the condition is not verified.

Example

```
(if (is Mover P1) Orthogonal Diagonal)
```

12
Range Functions

Range functions are ludemes that define a range of integer values with a lower and upper bound (inclusive). Ranges are useful for restricting integer values to sensible limits, e.g. for capping maximum bets in betting games or for situations in which negative values should be avoided.

12.1 Range

The base `range` function defines a range with upper and lower board (inclusive), optionally according to some specified condition.

12.1.1 range

Returns a range of values (inclusive) according to some specified condition.

Format

For a range between two int functions.

```
(range <int> <int>)
```

where:

- `<int>`: Lower extent of range (inclusive).
- `<int>`: Upper extent of range (inclusive).

For a range between two integers.

```
(range int int)
```

where:

- `int`: Lower extent of range (inclusive).
- `int`: Upper extent of range (inclusive).

Examples

```
(range (from) (to))

(range 1 9)
```

12.2 Math

Math range functions return a range based on specified inputs.

12.2.1 exact

Returns a range of exactly one value.

Format

```
(exact <int>)
```

where:

- `<int>`: The value in question.

Example

```
(exact 4)
```

Remarks

The exact value is both the minimum and maximum of its range.

12.2.2 max

Returns a range with a specified maximum (inclusive).

Format

```
(max <int>)
```

where:

- `<int>`: Upper extent of range (inclusive).

Example

```
(max 4)
```

12.2.3 min

Returns a range with a specified minimum (inclusive).

Format

```
(min <int>)
```

where:

- `<int>`: Lower extent of range (inclusive).

Example

```
(min 4)
```

13
Utilities

Utilities ludemes are useful support classes used by various other types of ludemes.

13.1 Directions

Direction utilities define the various types of directions used in game descriptions. These include:

- *absolute* directions that remain constant in all contexts, and
- *relative* directions that depend on the player and their orientation.

13.1.1 absoluteDirection

Describes categories of absolute directions.

Value	Description
All	All directions.
Angled	Angled directions.
Adjacent	Adjacent directions.
Axial	Axial directions.
Orthogonal	Orthogonal directions.
Diagonal	Diagonal directions.
Off	Off-diagonal directions.
SameLayer	Directions on the same layer.
Upward	Upward directions.

Downward	Downward directions.
Rotational	Rotational directions.
N	North.
E	East.
S	South.
W	West.
NE	North-East.
SE	South-East.
NW	North-West.
SW	South-West.
NNW	North-North-West.
WNW	West-North-West.
WSW	West-South-West.
SSW	South-South-West.
SSE	South-South-East.
ESE	East-South-East.
ENE	East-North-East.
NNE	North-North-East.
CW	Clockwise directions.
CCW	Counter-Clockwise directions.
In	Inwards directions.
Out	Outwards directions.
U	Upper direction.
UN	Upwards-North direction.
UNE	Upwards-North-East direction.
UE	Upwards-East direction.
USE	Upwards-South-East direction.
US	Upwards-South direction.
USW	Upwards-South-West direction.
UW	Upwards-West direction.
UNW	Upwards-North-West direction.
D	Down direction.
DN	Down-North direction.
DNE	Down-North-East.
DE	Down-East direction.
DSE	Down-South-East.
DS	Down-South direction.

DSW	Down-South-West.
DW	Down-West direction.
DNW	Down North West.

13.1.2 compassDirection

Intercardinal directions.

Value	Description
N	North.
NNE	North-North-East.
NE	North-East.
ENE	East-North-East.
E	East.
ESE	East-South-East.
SE	South-East.
SSE	South-South-East.
S	South.
SSW	South-South-West.
SW	South-West.
WSW	West-South-West.
W	West.
WNW	West-North-West.
NW	North-West.
NNW	North-North-West.

13.1.3 relativeDirection

Describes categories of relative directions.

Value	Description
Forward	Forward (only) direction.
Backward	Backward (only) direction.
Rightward	Rightward (only) direction.
Leftward	Leftward (only) direction.
Forwards	Forwards directions.
Backwards	Backwards directions.
Rightwards	Rightwards directions.
Leftwards	Leftwards directions.
FL	Forward-Left direction.

FLL	Forward-Left-Left direction.
FLLL	Forward-Left-Left-Left direction.
BL	Backward-Left direction.
BLL	Backward-Left-Left direction.
BLLL	Backward-Left-Left-Left direction.
FR	Forward-Right direction.
FRR	Forward-Right-Right direction.
FRRR	Forward-Right-Right-Right direction.
BR	Backward-Right direction.
BRR	Backward-Right-Right direction.
BRRR	Backward-Right-Right-Right direction.
SameDirection	Same direction.
OppositeDirection	Opposite direction.

13.2 End

This section describes support utility ludemes relevant to end rules.

13.2.1 score

Defines a score to set when using the `(byScore ...)` end rule.

Format

```
(score <roleType> <int>)
```

where:

- `<roleType>`: The role of the player.
- `<int>`: The score of the player.

Example

```
(score P1 100)
```

13.3 Equipment

Ludeme utilities are useful support classes that various types of ludeme classes refer to.

13.3.1 card

Defines an instance of a playing card.

Format

```
(card <cardType> rank:int value:int [trumpRank:int] [trumpValue:int]
     [biased:int])
```

where:

- `<cardType>`: The type of the card.
- `rank:int`: The rank of the card.
- `value:int`: The value of the card.
- `[trumpRank:int]`: The trump rank of the card.
- `[trumpValue:int]`: The trump value of the card.
- `[biased:int]`: The biased value of the card.

Example

```
(card Seven rank:0 value:0 trumpRank:0 trumpValue:0)
```

13.3.2 hint

Defines a hint value to a region or a specific site.

Format

For creating hints in a region.

```
(hint {int} [int])
```

where:

- `{int}`: The locations.
- `[int]`: The value of the hint [0].

For creating hint in a site.

```
(hint int [int])
```

where:

- `int`: The location.
- `[int]`: The value of the hint [0].

Examples

```
(hint {0 1 2 3 4 5} 1)

(hint 1 1)
```

Remarks

This is used only for deduction puzzles.

13.3.3 region

Defines a region of sites within a container.

13.3.4 values

Defines the set of values of a graph variable in a deduction puzzle.

Format

```
(values <siteType> <range>)
```

where:

- `<siteType>`: The graph element type.
- `<range>`: The range of the values.

Example

```
(values Cell (range 1 9))
```

13.4 Graph

Graph utilities are support classes for describing the graph that defines a game board.

13.4.1 graph

Defines the graph of a custom board described by a set of vertices and edges.

Format

```
(graph vertices:{{<float>}} [edges:{{int}}])
```

where:

- `vertices:{{<float>}}`: List of vertex positions in x y or x y z format.
- `[edges:{{int}}]`: List of vertex index pairs vi vj describing edge end points.

Example

```
(graph
    vertices:{ { 0 0} { 1.5 0 0.5 } { 0.5 1 } }
    edges:{ { 0 1} { 0 2 } { 1 2 } }
)
```

13.4.2 poly

Defines a polygon composed of a list of floating point (x,y) pairs.

Format

For building a polygon with float points.

```
(poly {{<float>}})
```

where:

- `{{<float>}}`: Float points defining polygon.

For building a polygon with DimFunction points.

```
(poly {{<dimFunction>}})
```

where:

- `{{<dimFunction>}}`: Float points defining polygon.

Examples

```
(poly { { 0 0} { 0 2.5 } { 4.75 1 } })

(poly { { 0 0} { 0 2.5 } { 4.75 1 } })
```

Remarks

The polygon can be concave.

13.5 Math

Math utilities are support classes for various numerical ludemes.

13.5.1 count

Associates an item with a count.

Format

```
(count <string> <int>)
```

where:

- `<string>`: Item description.
- `<int>`: Number of items.

Example

```
(count "Pawn1" 8)
```

Remarks

This ludeme is used for lists of items with counts, such as (placeRandom ...).

13.5.2 pair

Defines a pair of two integers, two strings or one integer and a string.

Format

For a pair of integers.

```
(pair int int)
```

where:

- `int`: The key of the pair.
- `int`: The corresponding value.

For a pair of a RoleType and an Integer.

```
(pair <roleType> int)
```

where:

- `<roleType>`: The key of the pair.
- `int`: The corresponding value.

For a pair of tow RoleTypes.

```
(pair <roleType> <roleType>)
```

where:

- `<roleType>`: The key of the pair.
- `<roleType>`: The corresponding value.

For a pair of two strings.

```
(pair <string> <string>)
```

where:

- `<string>`: The key of the pair.
- `<string>`: The corresponding value.

For a pair of an integer and a string.

```
(pair int <string>)
```

where:

- `int`: The key of the pair.
- `<string>`: The corresponding value.

For a pair of a RoleType and a string.

```
(pair <roleType> <string>)
```

where:

- `<roleType>`: The key of the pair.
- `<string>`: The corresponding value.

For a pair of a RoleType and an ordered graph element type.

```
(pair <roleType> <landmarkType>)
```

where:

- <roleType>: The key of the pair.
- <landmarkType>: The landmark of the value.

For a pair of a RoleType and a value.

```
(pair <string> <roleType>)
```

where:

- <string>: The key of the pair.
- <roleType>: The corresponding value.

Examples

```
(pair 5 10)

(pair P1 10)

(pair P1 P2)

(pair "A1" "C3")

(pair 0 "A1")

(pair P1 "A1")

(pair P1 LeftSite)

(pair "A1" P1)
```

Remarks

This is used for the map ludeme.

13.6 Moves

Moves utilities are support classes for defining and generating legal moves.

13.6.1 between

Gets all the conditions or effects related to the location between "from" and "to".

Format

```
(between [before:<int>] [<rangeFunction>] [after:<int>] [if:<boolean>]
     [trail:<int>] [<apply>])
```

where:

- `[before:<int>]`: Lead distance up to "between" section.
- `[<rangeFunction>]`: Range of the "between" section.
- `[after:<int>]`: Trailing distance after "between" section.
- `[if:<boolean>]`: The condition on the location.
- `[trail:<int>]`: The piece to let on the location.
- `[<apply>]`: Actions to apply.

Example

```
(between if:(is Enemy (who at:(between))) (apply (remove (between))))
```

13.6.2 flips

Sets the flips value of a piece.

Format

```
(flips int int)
```

where:

- `int`: The first value of the flip.
- `int`: The second value of the flip.

Example

```
(flips 1 2)
```

13.6.3 from

Specifies operations based on the "from" location.

Format

```
(from [<siteType>] ([<region>] | [<int>]) [level:<int>]
     [if:<boolean>])
```

where:

- `[<siteType>]`: The graph element type.
- `[<region>]`: The region of the "from" location.
- `[<int>]`: The "from" location.
- `[level:<int>]`: The level of the "from" location.
- `[if:<boolean>]`: The condition on the "from" location.

Example

```
(from (last To) level:(level))
```

13.6.4 piece

Specifies operations based on the "what" data.

Format

```
(piece (<string> | <int> | {<string>} | {<int>}) [state:<int>])
```

where:

- `<string>`: The name of the component.
- `<int>`: The index of the component [The component with the index corresponding to the index of the mover, (mover)].
- `{<string>}`: The names of the components.
- `{<int>}`: The indices of the components.

- [state:<int>]: The local state value to put on the site where the piece is placed.

Example

```
(piece (mover))
```

13.6.5 player

Specifies operations based on the "who" data.

Format

```
(player <int>)
```

where:

- <int>: The index of the player [(mover)].

Examples

```
(player (mover))

(player 2)
```

13.6.6 to

Specifies operations based on the "to" location.

Format

```
(to [<siteType>] ([<region>] | [<int>]) [level:<int>] [<rotations>]
     [if:<boolean>] [<apply>])
```

where:

- [<siteType>]: The graph element type.
- [<region>]: The region of "to" the location.
- [<int>]: The "to" location.
- [level:<int>]: The level of the "to" location.
- [<rotations>]: Rotations of the "to" location.
- [if:<boolean>]: The condition on the "to" location.

- [<apply>]: Effect to apply to the “to” location.

Example

```
(to (last To) level:(level))
```

14
Types

This package defines various types used to specify the behaviour of ludemes. Types are constant values denoted in `UpperCamelCase` syntax.

14.1 Board

Board types are constant values for specifying various aspects of the board and its constituent graph elements.

14.1.1 basisType

Defines known tiling types for boards.

Value	Description
NoBasis	No tiling; custom graph.
Triangular	Triangular tiling.
Square	Square tiling.
Hexagonal	Hexagonal tiling.
T33336	Semi-regular tiling made up of hexagons surrounded by triangles.
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.
T3464	Rhombitrihexahedral tiling (e.g. Kensington).
T3636	Semi-regular tiling 3.6.3.6 made up of hexagons with interstitial triangles.
T4612	Semi-regular tiling made up of squares, hexagons and dodecagons.

T488	Semi-regular tiling 4.8.8. made up of octagons with interstitial squares.
T31212	Semi-regular tiling made up of triangles and dodecagons.
T333333_33434	Tiling 3.3.3.3.3.3,3.3.4.3.4.
SquarePyramidal	Square pyramidal tiling (e.g. Shibumi).
HexagonalPyramida	Hexagonal pyramidal tiling.
Circle	Circular tiling (e.g. Round Merels).
Spiral	Spiral tiling (e.g. Mehen).
Dual	Tiling derived from the weak dual of a graph.
Brick	Brick tiling using 1x2 rectangular brick tiles.
Mesh	Mesh formed by random spread of points within an outline shape.
Morris	Morris tiling with concentric square rings and empty centre.
Celtic	Tiling on a square grid based on Celtic knotwork.
QuadHex	Quadhex board consisting of a hexagon tessellated by quadrilaterals (e.g. Three Player Chess).

14.1.2 landmarkType

Defines certain landmarks that can be used to specify individual sites on the board.

Value	Description
CentreSite	The central site of the board.
LeftSite	The site that is furthest to the left.
RightSite	The site that is furthest to the right
Topsite	The site that is furthest to the top.
BottomSite	The site that is furthest to the bottom.
FirstSite	The first site indexed in the graph.
LastSite	The last site indexed in the graph.

14.1.3 puzzleElementType

Defines the possible types of variables that can be used in deduction puzzles.

Value	Description
Cell	A variable corresponding to a cell.
Edge	A variable corresponding to an edge.
Vertex	A variable corresponding to a vertex.
Hint	A variable corresponding to a hint.

14.1.4 regionTypeDynamic

Defines regions which can change during play.

Value	Description
Empty	All the empty sites of the current state.
NotEmpty	All the occupied sites of the current state.
Own	All the sites occupied by a piece of the mover.
NotOwn	All the sites not occupied by a piece of the mover.
Enemy	All the sites occupied by a piece of an enemy of the mover.
NotEnemy	All the sites empty or occupied by a `Neutral` piece.
AllPlayers	All the sites occupied.

14.1.5 regionTypeStatic

Defines known (predefined) regions of the board.

Value	Description
Rows	Row areas.
Columns	Column areas.
AllDirections	All direction areas.
HintRegions	Hint areas.
Layers	Layers areas.
Diagonals	diagonal areas.
SubGrids	SubGrid areas.
Regions	Region areas.
Vertices	Vertex areas.
Corners	Corner areas.
Sides	Side areas.
SidesNoCorners	Side areas that are not corners.
AllSites	All site areas.
Touching	Touching areas.

14.1.6 relationType

Defines the possible relation types between graph elements.

Value	Description
Orthogonal	Orthogonal relation.
Diagonal	Diagonal relation.
Off	Diagonal-off relation.
Adjacent	Adjacent relation.
All	Any relation.

14.1.7 shapeType

Defines shape types for known board shapes.

Value	Description
NoShape	No defined board shape.
Custom	Custom board shape defined by the user.
Square	Square board shape.
Rectangle	Rectangular board shape.
Triangle	Triangular board shape.
Hexagon	Hexagonal board shape.
Cross	Cross board shape.
Diamond	Diamond board shape.
Prism	Diamond board shape extended vertically.
Quadrilateral	General quadrilateral board shape.
Rhombus	Rhombus board shape.
Wheel	Wheel board shape.
Circle	Circular board shape.
Spiral	Spiral board shape.
Wedge	Wedge shape of height N with 1 vertex at the top and 3 vertices on the bottom, for Alquerque boards.
Star	Multi-pointed star shape.
Limping	Alternating sides are staggered.
Polygon	Regular shape with sides of the same length.

14.1.8 siteType

Defines the element types that make up each graph.

Value	Description
Vertex	Graph vertex.
Edge	Graph edge.
Cell	Graph cell/face.

14.1.9 stepType

Defines possible “turtle steps” for describing walks through adjacent sites.

Value	Description
F	Forward a step.
B	Backward a step.

L	Turn left a step.
R	Turn right a step.

14.1.10 tilingBoardlessType

Defines supported tiling types for boardless games.

Value	Description
Square	Square tiling.
Triangular	Triangular tiling.
Hexagonal	Hexagonal tiling.

14.2 Component

Component types are constant values for specifying various aspects of components in the game. These can include pieces, cards, dice, and so on.

14.2.1 cardType

Defines possible rank values of cards.

Value	Description
Joker	Joker rank.
Ace	Ace rank.
Two	Two rank.
Three	Three rank.
Four	Four rank.
Five	Five rank.
Six	Six rank.
Seven	Seven rank.
Eight	Eight rank.
Nine	Nine rank.
Ten	Ten rank.
Jack	Jack rank.
Queen	Queen rank.
King	King rank.

14.2.2 dealableType

Specifies which types of components can be dealt.

Value	Description
Dominoes	Domino component.
Cards	Card component.

14.2.3 suitType

Defines the possible suit types of cards.

Value	Description
Clubs	Club suit.
Spades	Spade suit.
Diamonds	Diamond suit.
Hearts	Heart suit.

14.3 Play

Play types are constant values for specifying various aspects of play. These are typically to do with the "start", "play" and "end" rules.

14.3.1 modeType

Defines the possible modes of play.

Value	Description
Alternating	Players alternate making discrete moves.
Simultaneous	Players move at the same time.
Simulation	Simulation game

14.3.2 repetitionType

Defines the possible types of repetition that can occur in a game.

Value	Description
InTurn	State repeated within a turn.
InGame	State repeated within a game.
Positional	State repeated within a game (pieces on the board only).
Situational	State repeated within a game (including whose turn it is).

Infinite	State repeated within a game (with the same sequence of moves leading up to it).

14.3.3 resultType

Defines expected outcomes for each game.

Value	Description
Win	Somebody wins.
Loss	Somebody loses.
Draw	Nobody wins.
Tie	Everybody wins.
Abandon	Game abandoned, typically for being too long.
Crash	Game stopped due to run-time error.

14.3.4 roleType

Defines the possible role types of the players in a game.

Value	Description
Neutral	Neutral role, owned by nobody.
P1	Player 1.
P2	Player 2.
P3	Player 3.
P4	Player 4.
P5	Player 5.
P6	Player 6.
P7	Player 7.
P8	Player 8.
P9	Player 9.
P10	Player 10.
P11	Player 11.
P12	Player 12.
P13	Player 13.
P14	Player 14.
P15	Player 15.
P16	Player 16.
Team1	Team 1 (index 1).
Team2	Team 2 (index 2).
Team3	Team 3 (index 3).

Team4	Team 4 (index 4).
Team5	Team 5 (index 5).
Team6	Team 6 (index 6).
Team7	Team 7 (index 7).
Team8	Team 8 (index 8).
Team9	Team 9 (index 9).
Team10	Team 10 (index 10).
Team11	Team 11 (index 11).
Team12	Team 12 (index 12).
Team13	Team 13 (index 13).
Team14	Team 14 (index 14).
Team15	Team 15 (index 15).
Team16	Team 16 (index 16).
Each	Applies to each player (for iteration), e.g. same piece owned by each player
Shared	Shared role, shared by all players.
All	All players.
Any	Any players.
Mover	Player who is moving
Next	Player who is moving next turn
Prev	Player who moved on prior turn
NonMover	Players who are not moving
Enemy	Enemy player.
Ally	Ally player.
NonAlly	Non-ally player.
Partner	Partner player.
NonPartner	Non-partner player.
NonNeutral	Non-neutral player.
Player	Placeholder for iterator over all players, e.g. from end.ForEach.

14.3.5 whenType

Defines when to perform certain tests or actions within a game.

Value	Description
StartOfMove	Start of a move.
EndOfMove	End of a move.
StartOfTurn	Start of a turn.
EndOfTurn	End of a turn.

StartOfRound	Start of a round.
EndOfRound	End of a round.
StartOfPhase	Start of a phase.
EndOfPhase	End of a phase.
StartOfGame	Start of a game.
EndOfGame	End of a game.
StartOfMatch	Start of a match.
EndOfMatch	End of a match.
StartOfSession	Start of a session.
EndOfSession	End of a session.

Part II

Metadata

15
Info Metadata

Ludii *metadata* is additional information about each game that exists outside its core logic. Relevant metadata includes general game information, rendering hints, and AI hints to improve the playing experience.

15.1 Metadata

The `metadata` ludeme is a catch-call for all metadata items.

15.1.1 metadata

The metadata of a game.

Format

```
(metadata [<info>] [<graphics>] [<ai>])
```

where:

- `[<info>]`: The info metadata.
- `[<graphics>]`: The graphics metadata.
- `[<ai>]`: Metadata for AIs playing this game.

Example

```
(metadata
    (info
        {
            (description "Description of The game")
            (source "Source of the game")
            (version "1.0.0")
            (classification "board/space/territory")
            (origin "Origin of the game.")
        }
    )
    (graphics
        {
            (board Style Go)
            (player Colour P1 (colour Black))
            (player Colour P2 (colour White))
        }
    )
    (ai (bestAgent "UCT"))
)
```

15.2 Info

The `info` metadata items.

15.2.1 info

General information about the game.

Format

```
(info (<infoItem> | {<infoItem>}))
```

where:

- `<infoItem>`: The info item of the game.
- {`<infoItem>`}: The info items of the game.

Example

```
(info
    {
        (description "Description of The game")
        (source "Source of
          the game")
        (version "1.0.0")
        (classification "board/space/territory")
        (origin "Origin of the game.")
    }
)
```

15.3 Info - Database

The "database" metadata items describe information about the game, which is automatically synchronised from the Ludii game database at https://ludii.games/library.php. All of the types listed in this section may be used for <infoItem> parameters in metadata.

15.3.1 aliases

Specifies a list of additional aliases for the game's name.

Format

```
(aliases {<string>})
```

where:

- {<string>}: Set of additional aliases for the name of this game.

Example

```
(aliases {"Caturanga" "Catur"})
```

15.3.2 author

Specifies the author of the game or ruleset.

Format

```
(author <string>)
```

where:

- <string>: The author of the game.

Example

```
(author "John Doe")
```

15.3.3 classification

Specifies the location of this game within the Ludii classification scheme.

Format

```
(classification <string>)
```

where:

- `<string>`: The game's location within the Ludii classification scheme.

Example

```
(classification "games/board/war/chess")
```

Remarks

The Ludii classification is a combination of the schemes used in H. J. R. Murray's *A History of Board Games other than Chess* and David Parlett's *The Oxford History of Board Games*, with additional categories to reflect the wider range of games supported by Ludii.

15.3.4 credit

Specifies the author of the .lud file and any relevant credit information.

Format

```
(credit <string>)
```

where:

- `<string>`: The author of the .lud file.

Example

```
(credit "A. Fool, April Fool Games, 1/4/2020")
```

Remarks

The is *not* for the author of the game or ruleset. The ”author” info item should be used for that.

15.3.5 date

Specifies the (approximate) date that the game was created.

Format

```
(date <string>)
```

where:

- <string>: The date the game was created.

Example

```
(date "2015-10-05")
```

Remarks

Date is specified in the format (YYYY-MM-DD).

15.3.6 description

Specifies a description of the game.

Format

```
(description <string>)
```

where:

- <string>: An English description of the game.

Example

```
(description "A traditional game that comes from Egypt.")
```

15.3.7 origin

Specifies the location of the earliest known origin for this game.

Format

```
(origin <string>)
```

where:

- <string>: Earliest known origin for this game.

Example

```
(origin "1953")
```

15.3.8 publisher

Specifies the publisher of the game.

Format

```
(publisher <string>)
```

where:

- `<string>`: The publisher of the game.

Example

```
(publisher "Games Inc.")
```

15.3.9 rules

Specifies an English description of the rules of a game.

Format

```
(rules <string>)
```

where:

- `<string>`: An English description of the game's rules.

Example

```
(rules
    "Try to make a line of four."
)
```

15.3.10 source

Specifies the reference for the game, or its currently chosen ruleset.

Format

```
(source <string>)
```

where:

- `<string>`: The source of the game's rules.

Example

```
(source "Murray 1969")
```

15.3.11 version

Specifies the latest Ludii version that this .lud is known to work for.

Format

```
(version <string>)
```

where:

- `<string>`: Ludii version in String form.

Example

```
(version "1.0.0")
```

Remarks

The version format is (Major version).(Minor version).(Build number). For example, the first major version for public release is ”1.0.0”.

16
Graphics Metadata

The `graphics` metadata items give hints for rendering the board and components, as well as custom UI behaviour, to customise the interface for specific games and improve the playing experience.

16.1 Board

The `(board ...)` ‘super’ metadata ludeme is used modify a graphic property of a board.

16.1.1 board

Sets a graphic data to the board.

Format

For setting the style of a board.

```
(board Style <containerStyleType>)
```

where:

- `<containerStyleType>`: Container style wanted for the board.

For setting the style Pen and Paper of a board.

```
(board Style PenAndPaper onlyEdges:<boolean>)
```

where:

- `onlyEdges:<boolean>`: True if only the edges have to be modified by the style.

For setting the thickness style.

```
(board StyleThickness <boardGraphicsType> <float>)
```

where:

- `<boardGraphicsType>`: The board graphics type to which the colour is to be applied (must be InnerEdge or OuterEdge).
- `<float>`: The assigned thickness scale for the specified boardGraphicsType.

For setting the board to be checkered.

```
(board Checkered [<boolean>])
```

where:

- `[<boolean>]`: Whether the graphic data should be applied or not [true].

For setting the background or the foreground of a board.

```
(board <pieceGroundType> image:<string> [fillColour:<colour>]
     [edgeColour:<colour>] [scale:<float>])
```

where:

- `<pieceGroundType>`: The type of data to apply to the board.
- `image:<string>`: Name of the image to draw.
- `[fillColour:<colour>]`: Colour for the inner sections of the image. Default value is the phase 0 colour of the board.
- `[edgeColour:<colour>]`: Colour for the edges of the image. Default value is the outer edge colour of the board.
- `[scale:<float>]`: Scale for the drawn image relative to the size of the board [1.0].

For setting the colour of the board.

```
(board Colour <boardGraphicsType> <colour>)
```

where:

- `<boardGraphicsType>`: The board graphics type to which the colour is to be applied.
- `<colour>`: The assigned colour for the specified boardGraphicsType.

For setting the shape of the board.

```
(board Shape <shapeType>)
```

where:

- `<shapeType>`: The shape of the board.

Examples

```
(board Style Chess)

(board Style Chess)

(board StyleThickness OuterEdges 2.0)

(board Checkered)

(board
    Background
    image:"octagon"
    fillColour:(colour White)
    edgeColour:(colour White)
    scale:1.2
)

(board Colour Phase2 (colour Cyan))

(board Shape Square)
```

16.2 No

The `(no ...)` 'super' metadata ludeme is used to not show a graphic property.

16.2.1 no

Hides a graphic element.

Format

```
(no <noBooleanType> [<boolean>])
```

where:

- `<noBooleanType>`: The type of data.
- `[<boolean>]`: True if the graphic data has to be hidden [true].

Examples

```
(no Board)

(no Animation)

(no HandScale)

(no Curves)
```

16.3 Others

The "other" metadata items are used to modify the UI for a given game for more specific data.

16.3.1 adversarialPuzzle

Indicates whether the game is an adversarial puzzle.

Format

```
(adversarialPuzzle [<boolean>])
```

where:

- `[<boolean>]`: Whether the game is an adversarial puzzle or not [true].

Example

```
(adversarialPuzzle)
```

Remarks

Used in games which are expressed as a N-player game, but are actually puzzles, e.g. Chess puzzle.

16.3.2 hintType

Indicates whether the hints for the puzzle should be shown.

Format

```
(hintType <puzzleHintType>)
```

where:

- `<puzzleHintType>`: How hints should be shown.

Example

```
(hintType TopLeft)
```

16.3.3 stackType

Sets the stack design for a container.

Format

```
(stackType [<roleType>] [<string>] [int] ([sites:{int}] | [site:int])
     [<siteType>] <pieceStackType> [<float>])
```

where:

- `[<roleType>]`: Player whose index we want to match.
- `[<string>]`: Container name to match.
- `[int]`: Container index to match.
- `[sites:{int}]`: Draw image on all specified sites.
- `[site:int]`: Draw image on this site.
- `[<siteType>]`: The GraphElementType for the specified sites [Cell].
- `<pieceStackType>`: Stack type for this piece.
- `[<float>]`: Scaling factor [1.0].

Example

```
(stackType Ground)
```

Remarks

Different stack types that can be specified are defined in PieceStackType. For games such as Snakes and Ladders, Backgammon, Tower of Hanoi, card games, etc.

16.3.4 suitRanking

Indicates the ranking for card suits (lowest to highest).

Format

```
(suitRanking {<suitType>})
```

where:

- `{<suitType>}`: Ranking for card suits.

Example

```
(suitRanking { Spades Hearts Diamonds Clubs })
```

Remarks

Should be used only for card games.

16.4 Piece

The `(piece ...)` 'super' metadata ludeme is used to modify a graphic property of a piece.

16.4.1 piece

Sets a graphic data to the pieces.

Format

For setting the style of a piece.

```
(piece Style [<roleType>] [<string>] <componentStyleType>)
```

where:

- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `<componentStyleType>`: Component style wanted for this piece.

For setting the name of a piece.

```
(piece <pieceNameType> [<roleType>] [piece:<string>] [state:int]
    [<string>])
```

where:

- `<pieceNameType>`: The type of data to apply to the pieces.
- `[<roleType>]`: Player whose index is to be matched.
- `[piece:<string>]`: Base piece name to match.
- `[state:int]`: State to match.
- `[<string>]`: Text to use.

For setting the families of the pieces.

```
(piece Families {<string>})
```

where:

- `{<string>}`: Set of family names for the pieces used in the game.

For setting the background or the foreground of a piece.

```
(piece <pieceGroundType> [<roleType>] [<string>] [state:int]
     image:<string> [fillColour:<colour>] [edgeColour:<colour>]
     [scale:<float>])
```

where:

- `<pieceGroundType>`: The type of data to apply to the pieces.
- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `[state:int]`: State to match.
- `image:<string>`: Name of the image to draw.
- `[fillColour:<colour>]`: Colour for the inner sections of the image. Default value is the fill colour of the component.
- `[edgeColour:<colour>]`: Colour for the edges of the image. Default value is the edge colour of the component.
- `[scale:<float>]`: Scale for the drawn image relative to the cell size of the container [1.0].

For setting the colour of a piece.

```
(piece Colour [<roleType>] [<string>] [state:int]
     [fillColour:<colour>] [strokeColour:<colour>]
     [secondaryColour:<colour>])
```

where:

- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `[state:int]`: State to match.
- `[fillColour:<colour>]`: Fill colour for this piece.
- `[strokeColour:<colour>]`: Stroke colour for this piece.
- `[secondaryColour:<colour>]`: Secondary colour for this piece.

For reflecting the piece.

```
(piece Reflect [<roleType>] [<string>] [vertical:<boolean>]
     [horizontal:<boolean>])
```

where:

- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `[vertical:<boolean>]`: Reflect image vertically.

- `[horizontal:<boolean>]`: Reflect image horizontally.

For rotating the piece.

`(piece Rotate [<roleType>] [<string>] degrees:int)`

where:

- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `degrees:int`: Degrees to rotate clockwise.

For scaling a piece.

`(piece Scale [<roleType>] [<string>] <float> [<float>])`

where:

- `[<roleType>]`: Player whose index is to be matched.
- `[<string>]`: Base piece name to match.
- `<float>`: Scaling factor in x direction.
- `[<float>]`: Scaling factor in y direction [same as x factor].

For scaling a piece by its value.

`(piece Scale ByValue [<boolean>])`

where:

- `[<boolean>]`: Whether the pieces should be scaled [true].

Examples

```
(piece Style ExtendedShogi)

(piece Rename piece:"Die" "Triangle")

(piece ExtendName P2 "2")

(piece AddStateToName)

(piece Families {"Defined" "Microsoft" "Pragmata" "Symbola"})

(piece Foreground "Pawn" image:"Pawn" fillColour:(colour White) scale:0.9)

(piece
    Background
    "Han"
    image:"octagon"
    fillColour:(colour White)
    edgeColour:(colour White)
)

(piece Colour P2 "CounterStar" fillColour:(colour Red))

(piece Reflect P2 vertical:true horizontal:true)

(piece Rotate P2 degrees:90)

(piece Scale "Pawn" .5)

(piece Scale "Disc" 1 .5)

(piece Scale ByValue)
```

16.5 Player

The "player" metadata items describe relevant player settings.

16.5.1 player

Sets a graphic element to a player.

Format

```
(player Colour <roleType> <colour>)
```

where:

- <roleType>: Player whose index is to be matched.
- <colour>: Colour wanted for this player.

Example

```
(player Colour P1 (colour Black))
```

16.6 Region

The (region ...) 'super' metadata ludeme is used to modify a graphic property of a region.

16.6.1 region

Sets a graphic element to a region.

Format

```
(region Colour [<string>] [<roleType>] [<siteType>] ([{int}] | [int])
     [<boardGraphicsType>] [<colour>])
```

where:

- [<string>]: Region to be coloured.
- [<roleType>]: Player whose index is to be matched (only for Region).
- [<siteType>]: The GraphElementType for the specified sites [Cell].
- [{int}]: Sites to be coloured.
- [int]: Site to be coloured.
- [<boardGraphicsType>]: Only apply colouring onto sites that are also part of this BoardGraphicsType.
- [<colour>]: The assigned colour for the specified boardGraphicsType.

Example

```
(region Colour "Home" Edge InnerEdges (colour Black))
```

16.7 Show

The `(show ...)` 'super' metadata ludeme is used to show a graphic property or an information during the game.

16.7.1 show

Shows a graphic property or an information.

Format

For showing properties on the sites.

```
(show <showSiteType> <showSiteDataType> [<shapeType>] [<boolean>])
```

where:

- `<showSiteType>`: The type of data to show.
- `<showSiteDataType>`: The type of data to apply to the sites.
- `[<shapeType>]`: The shape of the board's cells.
- `[<boolean>]`: Whether the data has to be applied [true].

For showing symbols on sites.

```
(show Symbol <string> [<string>] [<roleType>] [<siteType>]
     ([{int}] | [int]) [<boardGraphicsType>] [fillColour:<colour>]
     [edgeColour:<colour>] [scale:<float>] [rotation:int])
```

where:

- `<string>`: Name of the image to show.
- `[<string>]`: Draw image on all sites in this region.
- `[<roleType>]`: Player whose index is to be matched (only for Region).
- `[<siteType>]`: The GraphElementType for the specified sites [Cell].
- `[{int}]`: Draw image on all specified sites.
- `[int]`: Draw image on this site.
- `[<boardGraphicsType>]`: Only apply image onto sites that are also part of this BoardGraphicsType.
- `[fillColour:<colour>]`: Colour for the inner sections of the image. Default value is the fill colour of the component.
- `[edgeColour:<colour>]`: Colour for the edges of the image. Default value is the edge colour of the component.
- `[scale:<float>]`: Scale for the drawn image relative to the cell size of the container [1.0].

- [rotation:int]: The rotation of the symbol.

For showing symbols on sites.

```
(show Line {{int}} [<colour>] [scale:<float>])
```

where:

- {{int}}: The line to draw (pairs of vertices).
- [<colour>]: The colour of the line.
- [scale:<float>]: The scale of the line.

For showing properties to edges.

```
(show Edges [<edgeType>] [<relationType>] [connection:<boolean>]
     [<lineStyle>] [<colour>])
```

where:

- [<edgeType>]: EdgeType condition [All].
- [<relationType>]: RelationType condition[Neighbour].
- [connection:<boolean>]: If this concerns cell connections, rather than graph edges [false].
- [<lineStyle>]: Line style for drawing edges [ThinDotted].
- [<colour>]: Colour in which to draw edges [LightGrey].

For showing properties.

```
(show <showBooleanType> [<boolean>])
```

where:

- <showBooleanType>: The type of data to show.
- [<boolean>]: Whether the graphic data has to be showed. [true].

For showing properties on a piece.

```
(show Piece <showComponentDataType> [<roleType>] [<string>]
     [<valueLocationType>])
```

where:

- <showComponentDataType>: The type of data to show.
- [<roleType>]: Player whose index is to be matched.

- [<string>]: Base piece name to match.
- [<valueLocationType>]: The location to draw the value [Corner].

For showing the check message.

```
(show Check [<roleType>] [<string>])
```

where:

- [<roleType>]: Player whose index is to be matched.
- [<string>]: Base piece name to match.

For showing the score.

```
(show Score [<whenScoreType>])
```

where:

- [<whenScoreType>]: When the score should be shown [Always].

Examples

```
(show Cell Shape Square)

(show Sites AsHoles)

(show Symbol "water" Cell 15 scale:0.85)

(show Line { { 1 0} { 2 4 } })

(show Edges Diagonal Thin)

(show Pits)

(show PlayerHoles)

(show RegionOwner)

(show Piece State)

(show Piece Value)

(show Check "King")

(show Score Never)
```

16.8 Util

The "util" metadata items are used for setting miscellaneous properties of the current game.

16.8.1 boardGraphicsType

Value	Description
InnerEdges	Edges that are not along a board side.
OuterEdges	Edges that define a board side.
Phase0	Cells in phase 0, e.g. dark cells on the Chess board.
Phase1	Cells in phase 1, e.g. light cells on the Chess board.
Phase2	Cells in phase 2, e.g. for hexagonal tiling colouring.
Phase3	Cells in phase 3, e.g. for exotic tiling colourings.
Symbols	Symbols drawn on the board, e.g. Senet, Hnefatafl, Royal Game of Ur...
Vertices	Intersections of lines on the board, e.g. where Go stones are played.

16.8.2 componentStyleType

Supported style types for rendering particular components.

Value	Description
Piece	Style for pieces.
Tile	Style for tiles (components that fill a cell and may have marked paths).
Card	Style for playing cards.
Die	Style for die components used as playing pieces.
Domino	Style for dominoes
LargePiece	Style for large pieces that srtaddle more than once site, e.g. the L Game.
ExtendedShogi	Extended style for Shogi pieces.
ExtendedXiangqi	Extended style for Shogi pieces.

16.8.3 containerStyleType

Supported style types for rendering particular boards.

Value	Description
Board	General style for boards.
Hand	General style for player hands.
Deck	General style for player Decks.
Dice	General style for player Dice.
Boardless	General style for boardless games, e.g. Andantino.

ConnectiveGoal	General board style for games with connective goals.
Mancala	General style for Mancala boards.
PenAndPaper	General style for the pen & paper style games, such as graph games.
Shibumi	Style for square pyramidal games played on the Shibumi board, e.g. Spline.
Spiral	General style for games played on a spiral board, e.g. Mehen.
Isometric	General style for games played on a isometric board.
Puzzle	General style for deduction puzzle boards.
Agon	Custom style for the Agon board.
Backgammon	Custom style for the Backgammon board.
Chess	Custom style for the Chess board.
ChineseCheckers	Custom style for the Chinese Checkers board.
Connect4	Custom style for the Connect4 board.
Goose	Custom style for the Game of the Goose board.
Go	Custom style for the Go board.
Graph	General style for graph game boards.
HoundsAndJackals	Custom style for the Hounds and Jackals (58 Holes) board.
Janggi	Custom style for the Janggi board.
Lasca	Custom style for the Lasca board.
Pachisi	Custom style for the Pachisi board.
Ploy	Custom style for the Ploy board.
Scripta	Custom style for the XII Scripta board.
Shogi	Custom style for the Shogi board.
SnakesAndLadders	Custom style for the Snakes and Ladders board.
Surakarta	Custom style for the Surakarta board.
Tafl	Custom style for Tafl boards.
Xiangqi	Custom style for the Xiangqi board.
UltimateTicTacToe	Custom style for the Ultimate Tic-Tac-Toe board.
Futoshiki	Custom style for the Futoskiki puzzle board.
Hashi	Custom style for the Hashi puzzle board.
Kakuro	Custom style for the Kakuro puzzle board.
Sudoku	Custom style for the Sudoku board.

16.8.4 controllerType

Defines supported controller types for handling user interactions for particular topologies.

Value	Description
BasicController	Basic user interaction controller.

PyramidalController	User interaction controller for games played on pyramidal topologies.

16.8.5 edgeType

Defines edge type for drawing board elements, e.g. for graph games.

Value	Description
All	All board edges.
Inner	Inner board edges.
Outer	Outer board edges.
Interlayer	Interlayer board edges.

16.8.6 lineStyle

Defines line styles for drawing board elements, e.g. edges for graph games.

Value	Description
Thin	Thin line.
Thick	Thick line.
ThinDotted	Thin dotted line.
ThickDotted	Thick dotted line.
ThinDashed	Thin dashed line.
ThickDashed	Thick dashed line.
Hidden	Line not drawn.

16.8.7 pieceStackType

Defines different ways of visualising stacks of pieces.

Value	Description
Default	Stacked one above the other (with offset).
Ground	Spread on the ground, e.g. Snakes and Ladders or Pachisi.
Reverse	Reverse stacking downwards.
Fan	Spread to show each component like a hand of cards.
FanAlternating	Spread to show each component like a hand of cards, alternating left and right side of centre.
None	No visible stacking.
Backgammon	Stacked Backgammon-style in lines of five.
Count	Show just top piece, with the stack value as number.
Ring	Stacked Ring-style around cell perimeter.

16.8.8 puzzleHintType

Defines different ways of visualising stacks of pieces.

Value	Description
Default	Stacked one above the other (with offset).
None	Spread on the ground, e.g. Snakes and Ladders or Pachisi.
TopLeft	Reverse stacking downwards.

16.8.9 valueLocationType

Specifed where to draw state of an item in the interface, relative to its position.

Value	Description
None	No location.
Corner	At the top left corner of the item's location.
Middle	Centred on the item's location.

16.8.10 whenScoreType

Specifies when to show player scores to the user.

Value	Description
Always	Always show player scores.
Never	Never show player scores.
AtEnd	Only show player scores at end of game.

16.9 Util - Colour

The “colour” metadata items allow the user to specify preferred colours for use in other metadata items.

16.9.1 colour

Defines a colour for use in metadata items.

Format

For defining a colour with the Red Green Blue values.

```
(colour int int int)
```

where:

- `int`: Red component [0..255].
- `int`: Green component [0..255].
- `int`: Blue component [0..255].

For defining a colour with the Red Green Blue values and an alpha value.

```
(colour int int int int)
```

where:

- `int`: Red component [0..255].
- `int`: Green component [0..255].
- `int`: Blue component [0..255].
- `int`: Alpha component [0..255].

For defining a colour with the hex code.

```
(colour <string>)
```

where:

- `<string>`: Six digit hexadecimal code.

For defining a predefined colour.

```
(colour <userColourType>)
```

where:

- <userColourType>: Predefined user colour type.

Examples

```
(colour 255 0 0)

(colour 255 0 0 127)

(colour "#00ff1a")

(colour DarkBlue)
```

16.9.2 userColourType

Specifies the colour of the user.

Value	Description
White	Plain white.
Black	Plain black.
Grey	Medium grey.
LightGrey	Light grey.
VeryLightGrey	Very light grey.
DarkGrey	Dark grey.
VeryDarkGrey	Very dark grey.
Dark	Almost black.
Red	Plain red.
Green	Plain green.
Blue	Blue.
Yellow	Yellow.
Pink	Pink.
Cyan	Cyan.
Brown	Medium brown.
DarkBrown	Dark brown.
Purple	Purple.
Magenta	Magenta.
Turquoise	Torquoise.
Orange	Medium orange.
DarkOrange	Dark orange.
LightRed	Light red.

DarkRed	Dark red.
LightGreen	Light green.
DarkGreen	Dark green.
LightBlue	Light blue.
VeryLightBlue	Very light blue.
DarkBlue	Dark blue.
IceBlue	Light icy blue.
Gold	Gold.
Silver	Silver.
Bronze	Bronze.
GunMetal	Gun metal blue.
HumanLight	Light human skin tone.
HumanDark	Dark human skin tone.
Cream	Cream.
DeepPurple	Deep purple.
PinkFloyd	Pink.
BlackSabbath	Very dark bluish black.
KingCrimson	King of the crimsons.
MoodyBlues	Moody blues.
TangerineDream	Tangerine.

17
AI Metadata

Ludii's *artificial intelligence* (AI) agents use hints provided in the `ai` metadata items to help them play each game effectively. These AI hints can apply to the game as a whole, or be targeted at particular variant rulesets or combinations of options within each game. Games benefit from the AI metadata but do not depend upon it; that is, Ludii's default AI agents will still play each game if no AI metadata is provided, but probably not as well.

17.1 AI

The `ai` metadata category collects relevant AI-related information. This information includes which *search algorithm* to use for move planning, what its settings should be, which *heuristics* are most useful, and which *features* (i.e. geometric piece patterns) are most important.

17.1.1 ai

Defines metadata that can help AIs in the Ludii app to play this game at a stronger level.

Format

```
(ai [<bestAgent>] [<heuristics>] [<features>])
```

where:

- `[<bestAgent>]`: Can be used to specify the agent that is expected to perform best in this game. This algorithm will be used when the "Ludii AI" option is selected in the Ludii app.
- `[<heuristics>]`: Heuristics to be used by Alpha-Beta agents. If not specified, Alpha-Beta agents will default to a combination of Material and Mobility heuristics.
- `[<features>]`: Feature sets to be used for biasing MCTS-based agents. If not specified, Biased MCTS will not be available as an AI for this game in Ludii.

Example

```
(ai (bestAgent "UCT"))
```

Remarks

Specifying AI metadata for games is not mandatory.

17.2 Features

The `features` package includes information about features used to bias Monte Carlo playouts. Each feature describes a geometric pattern of pieces that is relevant to the game, and recommends moves to make – or to not make! – based on the current game state. Biasing random playouts to encourage good moves that intelligent humans would make, and discourage moves that they would not make, leads to more realistic playouts and stronger AI play.

For example, a game in which players aim to make line of 5 of their pieces might benefit from a feature that encourages the formation of open-ended lines of 4. Each feature represents a simple strategy relevant to the game.

17.2.1 features

Describes one or more sets of features (local, geometric patterns) to be used by Biased MCTS agents.

Format

For just a single feature set shared among players.

`(features [<featureSet>])`

where:

- `[<featureSet>]`: A single feature set.

For multiple feature sets (one per player).

`(features [{<featureSet>}])`

where:

- `[{<featureSet>}]`: A sequence of multiple feature sets (typically each applying to a different player).

Examples

```
(features
    (featureSet
        All
        { (pair "rel:to=<{}>:pat=<refl=true,rots=all,els=[-{}]>" 1.0) }
    )
)

(features
    {
        (featureSet P1 { (pair "rel:to=<{}>:pat=<els=[-{}]>" 1.0) })
        (featureSet P2 { (pair "rel:to=<{}>:pat=<els=[-{}]>" -1.0) })
    }
)
```

Remarks

The basic format of these features is described in: Browne, C., Soemers, D. J. N. J., and Piette, E. (2019). "Strategic features for general games." In Proceedings of the 2nd Workshop on Knowledge Extraction from Games (KEG) (pp. 70–75).

17.2.2 featureSet

Defines a single feature set, which may be applicable to either a single specific player in a game, or to all players in a game.

Format

```
(featureSet <roleType> {<pair>})
```

where:

- `<roleType>`: The Player (P1, P2, etc.) for which the feature set should apply, or All if it is applicable to all features in a game.
- {`<pair>`}: Complete list of all features and weights for this feature set.

Examples

```
(featureSet All { (pair "rel:to=<{}>:pat=<els=[-{}]>" 1.0) })

(featureSet P1 { (pair "rel:to=<{}>:pat=<els=[-{}]>" 1.0) })
```

Remarks

Use `All` for feature sets that are applicable to all players in a game, or `P1`, `P2`, etc. for feature sets that are applicable only to individual players.

17.3 Heuristics

The `heuristics` package includes information about which heuristics are relevant to the current game, and their optimal settings and weights. Heuristics are simple rules of thumb for estimating the positional strength of players in a given game state. Each heuristic focusses on a particular aspect of the game, e.g. material piece count, piece mobility, etc.

17.3.1 heuristics

Defines a collection of heuristics, which can be used by Alpha-Beta agent in Ludii for their heuristics state evaluations.

Format

F.

```
(heuristics [<heuristicTerm>])
```

where:

- `[<heuristicTerm>]`: A single heuristic term.

For a collection of multiple heuristic terms.

```
(heuristics [{<heuristicTerm>}])
```

where:

- `[{<heuristicTerm>}]`: A sequence of multiple heuristic terms, which will all be linearly combined based on their weights.

Examples

```
(heuristics (score))

(heuristics { (material) (mobilitySimple weight:0.01) })
```

17.4 Heuristics - Terms

The `terms` package includes the actual heuristic metrics that can be applied and their settings.

17.4.1 centreProximity

Defines a heuristic term based on the proximity of pieces to the centre of a game's board.

Format

```
(centreProximity [transformation:<heuristicTransformation>]
     [weight:<float>] [pieceWeights:{<pair>}])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- `[pieceWeights:{<pair>}]`: Weights for different piece types. If no piece weights are specified at all, all piece types are given an equal weight of 1.0. If piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(centreProximity pieceWeights:{ (pair "Queen" 1.0) (pair "King" -1.0) })
```

17.4.2 cornerProximity

Defines a heuristic term based on the proximity of pieces to the corners of a game's board.

Format

```
(cornerProximity [transformation:<heuristicTransformation>]
     [weight:<float>] [pieceWeights:{<pair>}])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- `[pieceWeights:{<pair>}]`: Weights for different piece types. If no piece weights are

specified at all, all piece types are given an equal weight of 1.0. If piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(cornerProximity pieceWeights:{ (pair "Queen" -1.0) (pair "King" 1.0) })
```

17.4.3 currentMoverHeuristic

Defines a heuristic term that adds its weight only for the player whose turn it is in any given game state.

Format

```
(currentMoverHeuristic [transformation:<heuristicTransformation>]
     [weight:<float>])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.

Example

```
(currentMoverHeuristic weight:1.0)
```

17.4.4 lineCompletionHeuristic

Defines a heuristic state value based on a player's potential to complete lines up to a given target length. This mostly follows the description of the N-in-a-Row advisor as described on pages 82-84 of: "Browne, C.B. (2009) Automatic generation and evaluation of recombination games. PhD thesis, Queensland University of Technology".

Format

```
(lineCompletionHeuristic [transformation:<heuristicTransformation>]
     [weight:<float>] [targetLength:int])
```

where:

- [transformation:<heuristicTransformation>]: An optional transformation to be applied to any raw heuristic score outputs.
- [weight:<float>]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- [targetLength:int]: The target length for line completions. If not specified, we automatically determine a target length based on properties of the game rules or board.

Example

```
(lineCompletionHeuristic targetLength:3)
```

17.4.5 material

Defines a heuristic term based on the material that a player has on the board and in their hand.

Format

```
(material [transformation:<heuristicTransformation>] [weight:<float>]
     [pieceWeights:{<pair>}])
```

where:

- [transformation:<heuristicTransformation>]: An optional transformation to be applied to any raw heuristic score outputs.
- [weight:<float>]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- [pieceWeights:{<pair>}]: Weights for different piece types. If no piece weights are specified at all, all piece types are given an equal weight of 1.0. If piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(material pieceWeights:{ (pair "Pawn" 1.0) (pair "Bishop" 3.0) })
```

17.4.6 mobilitySimple

Defines a simple heuristic term that multiplies its weight by the number of moves that a player has in a current game state.

Format

```
(mobilitySimple [transformation:<heuristicTransformation>]
     [weight:<float>])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.

Example

```
(mobilitySimple weight:0.5)
```

Remarks

Always produces a score of 0 for players who are not the current mover.

17.4.7 ownRegionsCount

Defines a heuristic term based on the sum of all counts of sites in a player's owned regions.

Format

```
(ownRegionsCount [transformation:<heuristicTransformation>]
     [weight:<float>])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.

Example

```
(ownRegionsCount weight:1.0)
```

17.4.8 playerRegionsProximity

Defines a heuristic term based on the proximity of pieces to the regions owned by a particular player.

Format

```
(playerRegionsProximity [transformation:<heuristicTransformation>]
     [weight:<float>] player:int [pieceWeights:{<pair>}])
```

where:

- [transformation:<heuristicTransformation>]: An optional transformation to be applied to any raw heuristic score outputs.
- [weight:<float>]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- player:int: The player whose owned regions we compute proximity to.
- [pieceWeights:{<pair>}]: Weights for different piece types. If no piece weights are specified at all, all piece types are given an equal weight of 1.0. If piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(playerRegionsProximity player:2)
```

17.4.9 playerSiteMapCount

Defines a heuristic term that adds up the counts in sites corresponding to values in Maps where Player IDs (e.g. 1, 2, etc.) may be used as keys.

Format

```
(playerSiteMapCount [transformation:<heuristicTransformation>]
     [weight:<float>])
```

where:

- [transformation:<heuristicTransformation>]: An optional transformation to be applied to any raw heuristic score outputs.
- [weight:<float>]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.

Example

```
(playerSiteMapCount weight:1.0)
```

17.4.10 regionProximity

Defines a heuristic term based on the proximity of pieces to a particular region.

Format

```
(regionProximity [transformation:<heuristicTransformation>]
     [weight:<float>] region:int [pieceWeights:{<pair>}])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- `region:int`: Index of the region to which we wish to compute proximity.
- `[pieceWeights:{<pair>}]`: Weights for different piece types. If no piece weights are specified at all, all piece types are given an equal weight of 1.0. In piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(regionProximity weight:-1.0 region:0)
```

17.4.11 score

Defines a heuristic term based on a Player's score in a game.

Format

```
(score [transformation:<heuristicTransformation>] [weight:<float>])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple

terms. If not specified, a default weight of 1.0 is used.

Example

```
(score)
```

17.4.12 sidesProximity

Defines a heuristic term based on the proximity of pieces to the sides of a game's board.

Format

```
(sidesProximity [transformation:<heuristicTransformation>]
     [weight:<float>] [pieceWeights:{<pair>}])
```

where:

- `[transformation:<heuristicTransformation>]`: An optional transformation to be applied to any raw heuristic score outputs.
- `[weight:<float>]`: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- `[pieceWeights:{<pair>}]`: Weights for different piece types. If no piece weights are specified at all, all piece types are given an equal weight of 1.0. If piece weights are only specified for some piece types, all other piece types get a weight of 0.

Example

```
(sidesProximity weight:-1.0)
```

17.5 Heuristics - Transformations

The *transformations* metadata items specify how to normalise the heuristics results into a useful range.

17.5.1 divNumBoardSites

Transforms heuristic scores by dividing them by the number of sites in a game's board.

Format

```
(divNumBoardSites)
```

Example

```
(divNumBoardSites)
```

Remarks

Can be used to approximately standardise heuristic values across games with different board sizes.

17.5.2 divNumInitPlacement

Transforms heuristic scores by dividing them by the number of pieces placed in a game's initial game state.

Format

```
(divNumInitPlacement)
```

Example

```
(divNumInitPlacement)
```

Remarks

Can be used to approximately standardise heuristic values across games with different initial numbers of pieces.

17.5.3 logisticFunction

Transforms heuristic scores by applying the logistic function to them: $f(x) = \frac{1}{1+\exp(x)}$.

Format

```
(logisticFunction)
```

Example

```
(logisticFunction)
```

Remarks

This guarantees that all transformed heuristic scores will lie in $[0, 1]$. May map too many different values only to the limits of this interval in practice.

17.5.4 tanh

Transforms heuristic scores by applying the tanh to them: $f(x) = \tanh(x)$.

Format

```
(tanh)
```

Example

```
(tanh)
```

Remarks

This guarantees that all transformed heuristic scores will lie in $[-1, 1]$. May map too many different values only to the limits of this interval in practice.

17.6 Misc

The `misc` package includes miscellaneous items relevant to the AI settings.

17.6.1 bestAgent

Describes the name of an algorithm or agent that is typically expected to be the best-performing algorithm available in Ludii for this game.

Format

```
(bestAgent <string>)
```

where:

- `<string>`: The name of the (expected) best agent for this game.

Example

```
(bestAgent "UCT")
```

Remarks

Some examples of names that Ludii can currently recognise are "Random", "Flat MC", "Alpha-Beta", "UCT", "MC-GRAVE", and "Biased MCTS".

17.6.2 pair

Defines a pair of a String and a floating point value. Typically used in AI metadata to assign a numeric value (such as a heuristic score, or some other weight) to a specific piece name.

Format

```
(pair <string> <float>)
```

where:

- `<string>`: The String value.
- `<float>`: The floating point value.

Example

```
(pair "Pawn" 1.0)
```

Part III

Metalanguage Features

18 Defines

The `define` is a mechanism for replacing text in game descriptions with simple short labels, much like *macros* are used in programming languages. Defines are the first of several Ludii metalanguage features intended to make game descriptions clearer and more powerful.

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.

Defines can occur anywhere in the game description file – even within ludemes – but are typically at the top of the file, before the `game` reference, to impart some structure. Defines can be nested within other defines... but not themselves! The convention is to name each `define` with a single compound word in “UpperCamelCase” format.

18.1 Example

For example, the game description for Breakthrough contains the following `define`:

```
(define "ReachedTarget" (in (lastTo) (region Mover)) )
```

This `define` wraps up the concept the current mover’s last “to” move landing in the target region with the label “ReachedTarget”. The game’s *end* rule can then be simplified and clarified as follows:

```
(end (if ("ReachedTarget") (result Mover Win)))
```

This concept is used in many games, all of which can reuse this `define` to simplify and clarify their own descriptions.

18.2 Parameters

Ludii `define`s can be parameterised for greater flexibility. The parameters to be passed in to a define can take the following form:

```
keyword
(clause ...)
name:keyword
name:(clause ...)
```

Parameters are matched to *insertion points* of the form #*N* within the define, where *N* is the number of the parameter to be instantiated. For example, the following define:

```
(define "Outcome" (result #1 #2))
```

can be instantiated with any of the following calls:

```
("Outcome" Mover Win)
("Outcome" (next) Lose)
("Outcome" All Draw)
```

to give:

```
(result Mover Win)
(result (next) Lose)
(result All Draw)
```

Parameterised `define`s *must* be surrounded by brackets that enclose the label and its parameters when instantiated. Defines can contain arbitrary text, but should have balanced brackets, i.e. the same number of open and close brackets '(' for ')' and '{' for '}'. Non-parameterised `define`s do not need to be bracketed, but it is recommended to do so for consistency and readability. Both of the following formats are allowed but the first format is preferred:

```
(end (if ("ReachedTarget") (result Mover Win)))
(end (if  "ReachedTarget"  (result Mover Win)))
```

18.3 Null Parameters

Sometimes a `define` might be useful but its parameters do not match the current circumstance. In this case, a null parameter placeholder character ' ' may be passed instead of that parameter, which simply instantiates to nothing. Null parameters make `define`s even more powerful, by allowing the same `define` to be used in different ways by different games. For example, the following `define`:

```
(define "HopSequenceCapture"
    (hop
        (between #1 #2
            if:(isEnemy (who at:(between)))
            (apply (remove (between) #3))
```

```
            )
            (to if:(in (to) (empty)))
            (consequence
                (if (canMove
                        (hop
                            (from (lastTo))
                            (between #1 #2
                                if:(and (not (in (between) (sitesToClear)))
                                   (isEnemy (who at:(between))))
                                (apply (remove (between) #3))
                            )
                            (to if:(in (to) (empty)))
                        )
                    )
                    (moveAgain)
                )
            )
        )
    )
```

is called as follows in the game Coyote:

```
("HopSequenceCapture" ~ ~ at:EndOfTurn)
```

The first two parameters are null placeholders, so all occurrences of “#1” and “#2” in the “HopSequenceCapture” `define` will be instantiated with the empty string “”, while all occurrences of “#3” will be instantiated with “at:EndOfTurn”.

18.4 Known Defines

External `define`s called *known defines* can also be called within a game description simply by invoking their names (with suitable parameters). Each such known `define` must:

- be declared in a file with the same name as its label,
- have the file extension *.def, and
- be located in Ludii’s ”def” folder (or below it).

The list of known defines provided with the Ludii distribution is given in Appendix B.
In addition, each game will typically have a known `ai` metadata entry of the form:

```
(metadata
    ...
    (ai
        "Chess_ai"
    )
)
```

This is a reference to the known `ai` define that is automatically generated for each game, which stores the relevant AI settings for that game and its various options. Details of the `ai` metadata format are given in Chapter 17.

19
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.

19.1 Syntax

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 one or more named *arguments*. Option are described as follows:

```
(option "Heading <Tag> args:{ <argA> <argB> <argC> ... } {
    (item "Item X" <Xa> <Xb> <Xc> ... "Description of item X.")
    (item "Item Y" <Ya> <Yb> <Yc> ... "Description of item Y.")****
    (item "Item Z" <Za> <Zb> <Zc> ... "Description of item Z.")*
    ...
})
```

where:

- `"Category A"` is the category name.
- `Tag` is a tag used to locate the position in the game description where the option is to be instantiated.

- `argA/B/C` are named arguments for each item.
- `"Item X/Y/Z"` are the item names.
- `Xa, Xb, Xc` are the actual option arguments to instantiate.
- `"Description of item X/Y/Z."` describe each item in user friendly terms.

Option items are referenced in the game description by tag-argument pairs `<Tag:arg>`. For example, this option call in the game description:

```
(ludeme <Tag:argB>)
```

would be instantiated as follows if the user selects the menu item "Category A/Item Y":

```
(ludeme Yb)
```

19.2 Option Priority

A number of asterisks may optionally be appended to the end of each option item. The number of asterisks indicate that item's *priority* rating, with a higher number meaning higher priority.

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. If more than one item exists with the highest priority rating, then the first item listed with this rating is chosen. For example, "Item Y" would be the highest priority item for "Category A", with priority rating ****, and the default option to be instantiated in the absence of any user-selected options.

19.3 Example

The following example shows the option mechanism in action for the game of Hex. The game description assumes the existence of two option categories – `Board` and `Result` – with item arguments `size` and `type`, respectively.

```
(game "Hex"
    (players 2)
    (equipment {
        (board (rhombus <Board:size>))
        (piece "Ball" Each)
        (regions P1 { (sites Side NE) (sites Side SW) })
        (regions P2 { (sites Side NW) (sites Side SE) })
    })
    (rules
        (meta (swap) )
        (play (add (empty)))
        (end (if (isConnected Mover) (result Mover <Result:type>)))
    )
)
```

The two option categories are declared as follows. Note that the 11x11 option has the highest priority, while the 10x10, 14x14 and 17x17 options are next priority below. These are the most common sizes of Hex boards, so are the most interesting options for the user.

```
(option "Board Size" <Board> args:{ <size> } {
    (item   "3x3"  <3> "The game is played on a 3x3 board.")
    (item   "4x4"  <4> "The game is played on a 4x4 board.")
    (item   "5x5"  <5> "The game is played on a 5x5 board.")
    (item   "6x6"  <6> "The game is played on a 6x6 board.")
    (item   "7x7"  <7> "The game is played on a 7x7 board.")
    (item   "8x8"  <8> "The game is played on a 8x8 board.")
    (item   "9x9"  <9> "The game is played on a 9x9 board.")
    (item "10x10" <10> "The game is played on a 10x10 board.")*
    (item "11x11" <11> "The game is played on a 11x11 board.")****
    (item "12x12" <12> "The game is played on a 12x12 board.")
    (item "13x13" <13> "The game is played on a 13x13 board.")
    (item "14x14" <14> "The game is played on a 14x14 board.")*
    (item "15x15" <15> "The game is played on a 15x15 board.")
    (item "16x16" <16> "The game is played on a 16x16 board.")
    (item "17x17" <17> "The game is played on a 17x17 board.")*
    (item "18x18" <18> "The game is played on a 18x18 board.")
    (item "19x19" <19> "The game is played on a 19x19 board.")
})

(option "End Rules" <Result> args:{ <type> } {
    (item "Standard"  <Win>  "The first player to connect his two sides wins.")*
    (item "Misere"    <Loss> "The first player to connect his two sides loses.")
})
```

If the user selects the “Board Size/9x9” and “End Rules > Misere” menu item, then the game will be instantiated as follows during compilation, to give a *misère* version of the game on a 9×9 board:

```
(game "Hex"
    (players 2)
    (equipment {
        (board (rhombus 9))
        (piece "Ball" Each)
        (regions P1 { (sites Side NE) (sites Side SW) })
        (regions P2 { (sites Side NW) (sites Side SE) })
    })
    (rules
        (meta (swap) )
        (play (add (empty)))
        (end (if (isConnected Mover) (result Mover Loss)))
    )
)
```

20
Rulesets

In addition to the `option` mechanism described in the previous chapter, the Ludii game compiler also supports a `ruleset` mechanism that allows user to declare custom rule sets defined by combinations of options. User defined rulesets are of the form:

```
(rulesets {
    (ruleset "Ruleset/Name A" { "Option A/Item M" "Option B/Item N" ...})
    (ruleset "Ruleset/Name B" { "Option C/Item O" "Option D/Item P" ...})*
    ...
})
```

where:

- `"Ruleset/"` denotes this as a "Ruleset" menu item.
- `"Name A/B"` are unique user-specified names for each ruleset.
- `"Option A/Item M", "Option B/Item N", ...` are the actual options that make up this ruleset, as declared in their respective menu items.

Rulesets have a similar priority rating mechanism to options, i.e. rulesets with more asterisks appended to their declaration are deemed higher priority.

20.1 Example

The following example shows the ruleset mechanism in action for the game of Seega. This game description has a single option category – `Board` – with two item arguments `size` and `numInitPiece`.

```
(game "Seega"
    (players 2)
    (equipment { (board (square <Board:size>)) ... })
```

```
    (rules (start (place "Ball" "Hand" count:<Board:numPieces>)) ...)
)

(option "Board Size" <Board> args:{ <size> <numPieces>} {
    (item "5x5" <5> <12> "The game is played on a 5x5 board.")**
    (item "7x7" <7> <24> "The game is played on a 7x7 board.")
    (item "9x9" <9> <40> "The game is played on a 9x9 board.")
})

(rulesets {
    (ruleset "Ruleset/Khamsawee" { "Board Size/5x5" })*
    (ruleset "Ruleset/Sebawee"   { "Board Size/7x7" })
    (ruleset "Ruleset/Tisawee"   { "Board Size/9x9" })
})
```

If the user selects "Ruleset/Sebawee" from the menu, then its option "Board Size/7x7" will be instantiated to give:

```
(game "Seega"
    (players 2)
    (equipment { (board (square 7)) ... })
    (rules (start (place "Ball" "Hand" count:24)) ...)
)
```

If no user-selected ruleset is specified, then the game is compiled with the highest priority ruleset by default.

21
Ranges

In order to simplify the description of *ranges* of values, consecutive runs of numbers can be expressed in the form `<int>..<int>` in game descriptions. Ranges includes their limits. For example, the following ranges:

```
7..20
3..-3
```

will expand to these numbers:

```
7 8 9 10 11 12 13 14 15 16 17 18 19 20
3 2 1 0 -1 -2 -3
```

The following range in the game Dash Guti will expand as shown during compilation:

```
(place "Counter1" (region {0..9}))
(place "Counter1" (region {0 1 2 3 4 5 6 7 8 9}))
```

21.1 Smart Ranges

[FUTURE WORK]

It is possible to also specify ranges based on site coordinates in `String` form, e.g. `"A1".."A12"`. If both limits are co-axial then the range will expand consecutive sites along that axis between the specified limits, as follows:

```
{"A1".."A12"}
{"A1" "A2" "A3" "A4" "A5" "A6" "A7" "A8" "A9" "A10" "A11" "A12"'}
```

```
{"C2".."F2"}
{"C2" "D2" "E2" "F2"}
```

Otherwise, the range will expand to all sites within an area delimited by its limits:

```
{"B2".."D5"}
```

will expand to:

```
{"B2" "B3" "B4" "B5" "C2" "C3" "C4" "C5" "D2" "D3" "D4" "D5"}
```

22
Constants

A number of pre-defined constants can be used in game descriptions. These are instantiated with their actual values at compile time.

22.1 Off

Denotes an off-board position.

Internal value: -1

Example

```
(not (= (where "King" Next) Off))
```

22.2 End

Denotes the end of a track.

Internal value: -2

Example

```
(track "Track1" {0..5 7..12 25..20 18..13 End} P1 directed:true)
```

22.3 Undefined

Denotes a general "undefined" condition, for example if the game logic queries the value of a site that is out of range of the board.

Internal value: -1

Acknowledgements

This work is part of the *Digital Ludeme Project*, funded by €2m European Research Council (ERC) Consolidator Grant #771292 being run by Cameron Browne at Maastricht University's Department of Data Science and Knowledge Engineering over 2018–23.

The Ludii team consists of Cameron Browne (Principal Investigator), Éric Piette, Matthew Stephenson and Walter Christ (Postdoctoral Researchers) and Dennis Soemers (PhD Candidate). We thank Tahmina Begum and Wijnand Engelkes for contributions to this document.

European Research Council
Established by the European Commission

A
Image List

This Appendix lists the image files provided with the Ludii distribution. These image names can be used in .lud files to associate particular images with named pieces or board symbols. For pieces, the app tries to find the image whose file name is closest to the defined name, e.g. "QueenLeft" will match the image file "queen.svg". For board symbols, an exact name match is required.

Credits for images appear in the About dialog for games in which they are used.

Image list as of Ludii v1.0.7.

---

animals

bear.svg
bull.svg
camel-alt1.svg
camel-alt2.svg
camel-alt3.svg
camel.svg
cat-alt1.svg
cat.svg
chicken.svg
cow.svg
coyote-alt1.svg
coyote.svg
crab.svg
crocodile.svg
dog-alt1.svg
dog.svg
dove.svg
dragon.svg
duck-alt1.svg
duck.svg
eagle.svg
elephant-alt1.svg
elephant-alt2.svg
elephant.svg
fish.svg
fox.svg
goat-alt1.svg
goat-alt2.svg
goat-alt3.svg
goat.svg
goose.svg
hare-alt1.svg

hare-alt2.svg
hare.svg
hen.svg
horse-alt1.svg
horse.svg
jaguar.svg
lamb.svg
leopard.svg
lion-alt1.svg
lion.svg
lioness.svg
monkey-alt1.svg
monkey-alt2.svg
monkey.svg
mountainlion.svg
mouse.svg
ox.svg
panther.svg
penguin.svg
prawn.svg
puma.svg
rabbit-alt1.svg
rabbit.svg
rat.svg
rhino.svg
seal.svg
sheep.svg
snake-alt1.svg
snake.svg
tiger-alt1.svg
tiger-alt2.svg
tiger.svg
wolf.svg

cards

card-jack.svg
card-joker-old.svg
card-joker.svg
card-king.svg
card-queen.svg
card-suit-club.svg
card-suit-diamond.svg
card-suit-heart.svg
card-suit-spade.svg
cardBack.svg

checkers/isometric

counterstar_isometric.svg
counter_isometric.svg
doublecounter_isometric.svg

checkers/plain

counter.svg
counterstar.svg
doublecounter.svg

chess/microsoft

bishop_microsoft.svg
king_microsoft.svg
knight_microsoft.svg
pawn_microsoft.svg
queen_microsoft.svg
rook_microsoft.svg

chess/plain

bishop.svg
king.svg
knight.svg
pawn.svg
queen.svg
rook.svg

chess/pragmata

bishop_pragmata.svg
king_pragmata.svg
knight_pragmata.svg
pawn_pragmata.svg
queen_pragmata.svg
rook_pragmata.svg

chess/symbola

bishop_symbola.svg
king_symbola.svg
knight_symbola.svg
pawn_symbola.svg
queen_symbola.svg
rook_symbola.svg

faces

symbola_cool.svg
symbola_happy.svg
symbola_neutral.svg
symbola_pleased.svg
symbola_sad.svg
symbola_scared.svg
symbola_worried.svg

fairyChess

amazon.svg
bishop_noCross.svg
boat.svg
boat_alt1.svg
cannon.svg
chariot.svg
commoner.svg
ferz.svg
ferz_noCross.svg
flag.svg
fool.svg
giraffe.svg
king_noCross.svg
knight_bishop.svg
knight_king.svg
knight_queen.svg
knight_rook.svg
knight_rotated.svg
mann.svg
moon.svg
unicorn.svg
wazir.svg
zebra-neck.svg
zebra.svg

hands

hand0.svg
hand1.svg
hand2.svg
hand3.svg
hand4.svg
hand5.svg
paper.svg
rock.svg
scissors.svg

hieroglyphs

2human.svg
2human_knee.svg
3ankh.svg
3ankh_side.svg
3bird.svg
3nefer.svg
ankh_waset.svg
senetpiece.svg
senetpiece2.svg
water.svg

Janggi/traditional

Byeong.svg
Cha.svg
Cho.svg
Han.svg
Jol.svg
MaJanggi.svg
Po.svg
Sa.svg
Sang.svg

Janggi/western

byeong_western.svg
cha_western.svg
cho_western.svg
han_western.svg
jol_western.svg
maJanggi_western.svg
po_western.svg
sang_western.svg
sa_western.svg

letters

a.svg
b.svg
c.svg
d.svg
e.svg
f.svg
g.svg
h.svg
I.svg
j.svg

k.svg
l.svg
m.svg
n.svg
o.svg
p.svg
q.svg
r.svg
s.svg
t.svg
u.svg
v.svg
w.svg
x.svg
y.svg
z.svg

mahjong

BambooEight.svg
BambooFive.svg
BambooFour.svg
BambooNine.svg
BambooOne.svg
BambooSeven.svg
BambooSix.svg
BambooThree.svg
BambooTwo.svg
CharacterEight.svg
CharacterFive.svg
CharacterFour.svg
CharacterNine.svg
CharacterOne.svg
CharacterSeven.svg
CharacterSix.svg
CharacterThree.svg
CharacterTwo.svg
CircleEight.svg
CircleFour.svg
CircleNine.svg
CircleOne.svg
CircleSeven.svg
CircleSix.svg
CircleThree.svg
CircleTwo.svg
CricleFive.svg
DragonGreen.svg
DragonRed.svg
DragonWhite.svg
FlowerBamboo.svg
FlowerChrysanthemum.svg
FlowerOrchid.svg
FlowerPlum.svg
SeasonAutumn.svg
SeasonSpring.svg
SeasonSummer.svg
SeasonWinter.svg
TileBack.svg
TileJoker.svg
WindEast.svg
WindNorth.svg
WindSouth.svg
WindWest.svg

misc

bean.svg
bike.svg
boy.svg
bread.svg
castle.svg
corn.svg
door.svg
dot.svg
egyptLion.svg
flower.svg
flowerHalf1.svg
flowerHalf2.svg
human.svg
minotaur.svg
oldMan.svg
paddle.svg
pawn3d.svg
robot.svg
rubble.svg
stick.svg
theseus.svg
urpiece.svg
waves.svg

ploy

Commander.svg
LanceT.svg
LanceW.svg
LanceY.svg
ProbeBigV.svg
ProbeI.svg
ProbeMinV.svg
Shield.svg

salta

Salta1Dot.svg
Salta1Moon.svg
Salta1Star.svg
Salta2Dot.svg
Salta2Moon.svg
Salta2Star.svg
Salta3Dot.svg
Salta3Moon.svg
Salta3Star.svg
Salta4Dot.svg
Salta4Moon.svg
Salta4Star.svg
Salta5Dot.svg
Salta5Moon.svg
Salta5Star.svg

shapes

cross.svg
diamond.svg
disc.svg
discDouble.svg
discDoubleStick.svg
discFlat.svg
discStick.svg
hex.svg
hexE.svg
line.svg
none.svg
octagon.svg
pentagon.svg
pyramid.svg
rectangle.svg
square.svg
star.svg
starOutline.svg
thinCross.svg
triangle.svg

shogi

shogi_blank.svg

shogi/study

fuhyo_study.svg
ginsho_study.svg
hisha_study.svg
kakugyo_study.svg
keima_study.svg
kinsho_study.svg
kyosha_study.svg
narigin_study.svg
narikei_study.svg
narikyo_study.svg
osho1_study.svg
osho_study.svg
ryuma_study.svg
ryuo_study.svg
tokin_study.svg

shogi/traditional

fuhyo.svg
ginsho.svg
hisha.svg
kakugyo.svg
keima.svg
kinsho.svg
kyosha.svg
narigin.svg
narikei.svg
narikyo.svg
osho.svg
osho1.svg
ryuma.svg
ryuo.svg
tokin.svg

stickDice

oldMan0.svg
oldMan1.svg
oldWoman0.svg
oldWoman1.svg
youngMan0.svg
youngMan1.svg
youngWoman0.svg
youngWoman1.svg

stratego

bomb.svg
captain.svg
colonel.svg
flag.svg
general.svg
lieutenant.svg
major.svg
marshal.svg
miner.svg
scout.svg
sergeant.svg
spy.svg

tafl

jarl.svg
knotSquare.svg
knotTriangle.svg
thrall.svg

war

bow.svg
catapult.svg
crossbow.svg
knife.svg
scimitar.svg
smallSword.svg
sword.svg

xiangqi/extended

archer.svg
deputy general.svg
diplomat.svg
general blue.svg
general green.svg
general grey.svg
general magenta.svg
general orange.svg
general red.svg
general white.svg
officer.svg

xiangqi

symbol.svg
symbol_left.svg
symbol_right.svg

xiangqi/traditional

jiang.svg
jiang_black.svg
ju.svg
ju_black.svg
ma.svg
ma_black.svg
pao.svg
pao_black.svg
shi.svg
shi_black.svg
xiang.svg
xiang_black.svg
zu.svg
zu_black.svg

xiangqi/western

jiang_black_western.svg
jiang_western.svg
ju_black_western.svg
ju_western.svg
ma_black_western.svg
ma_western.svg
pao_black_western.svg
pao_western.svg
shi_black_western.svg
shi_western.svg
xiang_black_western.svg
xiang_western.svg
zu_black_western.svg
zu_western.svg

B
Known Defines

This Appendix lists the known `define` structures provided with the Ludii distribution, which are available for use by game authors. Known defines can be found in the “def” package area (or below) with file extension *.def. See Chapter 18 for details on the `define` syntax.

B.1 def/board

B.1.1 "LascaBoard"

Defines the original Lasca board.

Example

```
("LascaBoard")
```

```
(define "LascaBoard"
    (board
        (graph
            vertices:{ {0 0} {2 0} {4 0} {6 0} {1 1} {3 1} {5 1} {0 2} {2 2} {4 2}
            {6 2} {1 3} {3 3} {5 3} {0 4} {2 4} {4 4} {6 4} {1 5} {3 5}
            {5 5} {0 6} {2 6} {4 6} {6 6}
            }
            edges:{ {0 4} {1 4} {1 5} {2 5} {2 6} {3 6} {4 7} {4 8} {5 8} {5 9}
            {6 9} {6 10} {7 11} {8 11} {8 12} {9 12} {9 13} {10 13} {11 14}
            {11 15} {12 15} {12 16} {13 16} {13 17} {14 18} {15 18} {15 19}
            {16 19} {16 20} {17 20} {18 21} {18 22} {19 22} {19 23} {20 23}
            {20 24}
            }
        )
    )
)
```

B.1.2 "HoundsAndJackalsBoard"

Defines the standard 58 Holes or Hounds & Jackals board.
This board defines tracks for two players, hardcoded in a particular shape.

Example

```
("HoundsAndJackalsBoard")
```

```
(define "HoundsAndJackalsBoard"
    (board
        (graph
            vertices:{
            { 9 27} { 9 24} { 9 21} { 9 18} { 9 15} { 9 12} { 9  9} { 9  6}
            { 9  3} { 9  0} { 3  0} { 3  2} { 3  4} { 3  6} { 3  8} { 3 10}
            { 3 12} { 3 14} { 3 16} { 3 18} { 3 20} { 3 22} { 3 24} { 3 26}
            { 3 28} { 4 30} { 6 31} { 8 32} {10 33} {15 27} {15 24} {15 21}
```

```
            {15 18} {15 15} {15 12} {15  9} {15  6} {15  3} {15  0} {21  0}
            {21  2} {21  4} {21  6} {21  8} {21 10} {21 12} {21 14} {21 16}
            {21 18} {21 20} {21 22} {21 24} {21 26} {21 28} {20 30} {18 31}
            {16 32} {14 33} {12 33}
            }
            edges:{
            { 0  1} { 1  2} { 2  3} { 3  4} { 4  5} { 5  6} { 6  7} { 7  8}
            { 8  9} { 9 10} {10 11} {11 12} {12 13} {13 14} {14 15} {15 16}
            {16 17} {17 18} {18 19} {19 20} {20 21} {21 22} {22 23} {23 24}
            {24 25} {25 26} {26 27} {27 28} {28 58} {29 30} {30 31} {31 32}
            {32 33} {33 34} {34 35} {35 36} {36 37} {37 38} {38 39} {39 40}
            {40 41} {41 42} {42 43} {43 44} {44 45} {45 46} {46 47} {47 48}
            {48 49} {49 50} {50 51} {51 52} {52 53} {53 54} {54 55} {55 56}
            {56 57} {57 58}
            }
        )
        {
        (track "Track1" {59  0..28 58} P1 directed:true)
        (track "Track2" {60 29..58   } P2 directed:true)
        }
    )
)
```

B.2 def/conditions

B.2.1 “HandEmpty”

Checks if all the sites in a specific hand are empty.
#1 = The owner of the hand, can be a RoleType or the index of the owner.

Example

```
("HandEmpty" Mover)
```

```
(define "HandEmpty"
    (= 0 (count in:(sites Hand #1)))
)
```

B.2.2 “IsInCheck”

Checks if a specific piece is threatened by any other enemy piece.
#1 = The name of the piece (without the number).
#2 = The roleType of the owner of the piece.
#3 = The location where is the piece. This parameter is optional. If not specify the current location of the piece is used.

Example

```
("IsInCheck" "Osho" Next)
("IsInCheck" "King" Mover (to))
```

```
(define "IsInCheck"
    (isThreatened (id #1 #2) #3)
)
```

B.2.3 "SameTurn"

Checks if the previous mover is the same player than the current mover, consequently this is the same turn.

Example

```
("SameTurn")
```

```
(define "SameTurn"
    (isPrev Mover)
)
```

B.2.4 "NoPiece"

Checks if a player has currently no piece.
#1 = The roleType of the owner of the piece.

Example

```
("NoPiece" Mover)
```

```
(define "NoPiece"
    (= (count Pieces #1) 0)
)
```

B.3 def/rules

B.4 def/rules/play

B.5 def/rules/play/end

B.5.1 "NoMoves"

Checks if the next player has no legal moves and apply a specific result to the next player. This ludemeplex can be used only in an ending condition.
#1 = The result to apply.

Example

```
("NoMoves" Loss)
```

```
(define "NoMoves"
    (if (noMoves Next) (result Next #1))
)
```

B.6 def/rules/play/moves

B.6.1 "StepForwardToEmpty"

Defines a step move to the forward direction according to the facing direction of the piece in the 'from' location to an empty site.

Example

```
("StepForwardToEmpty")
```

```
(define "StepForwardToEmpty"
    (step
        Forward
        (to if:(in (to) (empty)))
    )
)
```

B.6.2 "HopOrthogonalSequenceCaptureAgain"

Defines a sequence of hop move in all the orthogonal directions over an enemy to an empty site from the last 'to' location of the previous move. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopOrthogonalSequenceCaptureAgain")
("HopOrthogonalSequenceCaptureAgain" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopOrthogonalSequenceCaptureAgain"
    (hop
        (from (lastTo))
        Orthogonal
        (between
            #1
            #2
            if:(and (not (in (between) (sitesToClear))) (isEnemy (who at:(between))))
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        Orthogonal
                        (between
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
            )
        )
    )
)
```

B.6.3 "HopCapture"

Defines a hop move in all the adjacent directions over an enemy to an empty site. The enemy piece is removed.

Example

```
("HopCapture")
```

```
(define "HopCapture"
    (hop
        (between
            if:(isEnemy (who at:(between)))
            (apply (remove (between)))
        )
        (to if:(in (to) (empty)))
    )
)
```

B.6.4 "HopSequenceCapture"

Defines a sequence of hop move in all the adjacent directions over an enemy to an empty site. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopSequenceCapture")
("HopSequenceCapture" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopSequenceCapture"
    (hop
        (between
            #1
            #2
            if:(isEnemy (who at:(between)))
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        (between
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
```

```
            )
        )
    )
)
```

B.6.5 "StepOrthogonalToEmpty"

Defines a step move to all the orthogonal directions to an empty site.

Example

```
("StepOrthogonalToEmpty")
```

```
(define "StepOrthogonalToEmpty"
    (step Orthogonal (to if:(in (to) (empty))))
)
```

B.6.6 "HopOrthogonalSequenceCapture"

Defines a sequence of hop move in all the orthogonal directions over an enemy to an empty site. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopOrthogonalSequenceCapture")
("HopOrthogonalSequenceCapture" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopOrthogonalSequenceCapture"
    (hop
        Orthogonal
        (between
            #1
            #2
            if:(isEnemy (who at:(between)))
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        Orthogonal
                        (between
```

```
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
            )
        )
    )
)
```

B.6.7 "StepToEmpty"

Defines a step move to all the adjacent directions to an empty site.

Example

```
("StepToEmpty")
```

```
(define "StepToEmpty"
    (step
        (to if:(in (to) (empty)))
    )
)
```

B.6.8 "HopSequenceCaptureAgain"

Defines a sequence of hop move in all the adjacent directions over an enemy to an empty site from the last 'to' location of the previous move. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopSequenceCaptureAgain")
("HopSequenceCaptureAgain" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopSequenceCaptureAgain"
    (hop
        (from (lastTo))
        (between
```

```
            #1
            #2
            if:(and (not (in (between) (sitesToClear))) (isEnemy (who at:(between))))
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        (between
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
            )
        )
    )
)
```

B.6.9 "HopDiagonalSequenceCaptureAgain"

Defines a sequence of hop move in all the diagonal directions over an enemy to an empty site from the last 'to' location of the previous move. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopDiagonalSequenceCaptureAgain")
("HopDiagonalSequenceCaptureAgain" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopDiagonalSequenceCaptureAgain"
    (hop
        (from (lastTo))
        Diagonal
        (between
            #1
            #2
            if:(and (not (in (between) (sitesToClear))) (isEnemy (who at:(between))))
```

```
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        Diagonal
                        (between
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
            )
        )
    )
)
```

B.6.10 "StepDiagonalToEmpty"

Defines a step move to all the diagonal directions to an empty site.

Example

```
("StepDiagonalToEmpty")
```

```
(define "StepDiagonalToEmpty"
    (step Diagonal (to if:(in (to) (empty))))
)
```

B.6.11 "HopDiagonalSequenceCapture"

Defines a sequence of hop move in all the diagonal directions over an enemy to an empty site. The enemy pieces are removed.
#1 = Maximum distance before the hop [0].
#2 = Maximum distance after the hop [0].
#3 = When to perform the capture (immediately or at the end of the turn) [immediately].

Example

```
("HopDiagonalSequenceCapture")
("HopDiagonalSequenceCapture" before:(count Dim) after:(count Dim) at:EndOfTurn)
```

```
(define "HopDiagonalSequenceCapture"
    (hop
        Diagonal
        (between
            #1
            #2
            if:(isEnemy (who at:(between)))
            (apply (remove (between) #3))
        )
        (to if:(in (to) (empty)))
        (consequence
            (if (canMove
                    (hop
                        (from (lastTo))
                        Diagonal
                        (between
                            #1
                            #2
                            if:(and (not (in (between) (sitesToClear)))
                                    (isEnemy (who at:(between))))
                            (apply (remove (between) #3))
                        )
                        (to if:(in (to) (empty)))
                    )
                )
                (moveAgain)
            )
        )
    )
)
```

B.6.12 "StepForwardsToEmpty"

Defines a step move to all the forwards directions according to the facing direction of the piece in the 'from' location to an empty site.

Example

```
("StepForwardsToEmpty")
```

```
(define "StepForwardsToEmpty"
    (step Forwards (to if:(in (to) (empty))))
```

```
)
```

B.7 def/walk

B.7.1 "KnightWalk"

Defines a graphics turtle walk to get the locations where can move a Chess knight.

Example

```
("KnightWalk")
```

```
(define "KnightWalk"
    { {F F R F} {F F L F} }
)
```

B.7.2 "TWalk"

Defines a graphics turtle walk to build a large T piece.

Example

```
("TWalk")
```

```
(define "TWalk"
    { {F F F L F R R F F} }
)
```

B.7.3 "DominoWalk"

Defines a graphics turtle walk to build a domino as a large piece.

Example

```
("DominoWalk")
```

```
(define "DominoWalk"
    { {F R F R F L F L F R F R F} }
)
```

B.7.4 “LWalk”

Defines a graphics turtle walk to build a large L piece.

Example

```
("LWalk")
```

```
(define "LWalk"
    { {L F R F F} {R F L F F} }
)
```

C
Ludii Grammar

This Appendix lists the complete Ludii grammar for the current Ludii version. The Ludii grammar is generated automatically from the hierarchy of Java classes that implement the ludemes described in this document, using the *class grammar* approach described in C. Browne "A Class Grammar for General Games", *Computers and Games (CG 2016)*, Springer, LNCS 10068, pp. 169–184.

Ludii game descriptions (*.lud files) *must* conform to this grammar, but note that conformance does not guarantee compilation. Many factors can stop game descriptions from compiling, such as attempting to access a component using an undefined name, attempting to modify board sites that do not exist, and so on.

C.1 Compilation

The steps for compiling a game according to the grammar, from a given *.lud game description to an executable Java `Game` object, are as follows:

1. *Expand*: The *raw* text *.lud game description is expanded according to the metalanguage features described in Part III (defines, options, rulesets, ranges, constants, etc.) to give an *expanded* text description of the game with current options and rulesets instantiated.
2. *Tokenise*: The expanded text description is then *tokenised* into a *symbolic expression* in the form of a tree of simple tokens.
3. *Parse*: The token tree is parsed according to the current Ludii grammar for correctness.
4. *Compile*: The names of tokens in the token tree are then matched with known ludeme Java classes and these are compiled with the specified arguments, if possible, to give a `Game` object.
5. *Create*: The `Game` object calls its `create()` method to perform relevant preparations, such as deciding on an appropriate `State` type, allocating required memory, initialising necessary variables, etc.

C.2 Listing

```
//------------------------------------------------------------------------------
// game

<game>     ::= (game string [<players>] [<mode>] [<equipment>] [<rules.rules>])
               | <match>

//------------------------------------------------------------------------------
// game.players

<players>  ::= (players int [<moves>]) | (players {<players.player>})
<players.player> ::= (player [string] [<directionFacing>] [<moves>])

//------------------------------------------------------------------------------
// game.rules.play.moves

<moves>    ::= <effect.add> | <addScore> | <allCombinations> | <logical.and> |
               <append> | <apply> | <attract> | <avoidStoredState> | <bet> |
               <claim> | <custodial> | <effect.deal> | <decision> |
               <directionCapture> | <do> | <effect> | <enclosed> |
               <firstMoveOnTrack> | <flip> | <foreach.forEach> | <fromTo> |
               <hop> | <logical.if> | <intervene> | <leap> | <max.max> |
               <move> | <moveAgain> | <nonDecision> | <note> | <operator> |
               <logical.or> | <pass> | <playCard> | <priority> | <promote> |
               <propose> | <push> | <rememberState> | <effect.remove> |
               <roll> | <satisfy> | <select> | <effect.set.set> | <shoot> |
               <slide> | <sow> | <effect.step> | <surround> | <swap.swap> |
               <take> | <trigger> | <vote>

//------------------------------------------------------------------------------
// game.rules.play.moves.decision

<decision> ::= <move>
<move>     ::= (move Slide [<moves.from>] [string] [<direction>] [<moves.between>] [<moves.to>] [<then>])
               |
               (move Shoot <moves.piece> [<moves.from>] [<absoluteDirection>] [<moves.to>] [<then>])
               | (move Select <moves.from> [<moves.to>] [<then>]) |
               (move <moveMessageType> (string | {string}) [<then>]) |
               (move Promote [<siteType>] [<int>] <moves.piece> [<moves.player>
               | <roleType>] [<then>]) |
               (move Remove [<siteType>] (<int> |
               <sites>) [at:<whenType>] [<then>]) |
               (move Set TrumpSuit (<int> |
               <intArray.math.difference>) [<then>]) |
               (move Set NextPlayer (<moves.player> |
               <ints>) [<then>]) |
               (move Set Direction [<moves.to>] [{<int>} |
               <int>] [previous:<boolean>] [next:<boolean>] [<then>])
               |
               (move Step [<moves.from>] [<direction>] <moves.to> [stack:boolean] [<then>])
               |
```

```
               (move <moveSiteType> [<moves.piece>] <moves.to> [stack:boolean] [<then>])
               | (move Bet (<moves.player> |
               <roleType>) <range> [<then>]) |
               (move <moves.from> <moves.to> [count:<int>] [copy:<boolean>] [stack:boolean] [<roleType>] [<th
               | (move <moveSimpleType> [<then>]) |
               (move Leap [<moves.from>] {{<stepType>}} [forward:<boolean>] [rotations:<boolean>] <moves.to>
               |
               (move Hop [<moves.from>] [<direction>] [<moves.between>] <moves.to> [stack:boolean] [<then>])
<moveMessageType> ::= Propose | Vote
<moveSimpleType> ::= Pass | PlayCard
<moveSiteType> ::= Add | Claim

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect

<effect>   ::= <effect.add> | <addScore> | <attract> | <avoidStoredState> |
               <bet> | <claim> | <custodial> | <effect.deal> |
               <directionCapture> | <do> | <enclosed> | <firstMoveOnTrack> |
               <flip> | <foreach.forEach> | <fromTo> | <hop> | <intervene> |
               <leap> | <max.max> | <moveAgain> | <note> | <pass> |
               <playCard> | <priority> | <promote> | <propose> | <push> |
               <rememberState> | <effect.remove> | <roll> | <satisfy> |
               <select> | <effect.set.set> | <shoot> | <slide> | <sow> |
               <effect.step> | <surround> | <swap.swap> | <take> | <trigger> |
               <vote>
<apply>    ::= (apply [if:<boolean>] [<nonDecision>])
<attract>  ::= (attract [<moves.from>] [<absoluteDirection>] [<then>])
<bet>      ::= (bet (<moves.player> | <roleType>) <range> [<then>])
<claim>    ::= (claim [<moves.piece>] <moves.to> [<then>])
<custodial> ::= (custodial [<moves.from>] [<absoluteDirection>] [<moves.between>] [<moves.to>] [<then>])
<directionCapture> ::= (directionCapture [<moves.from>] [<moves.to>] [opposite:<boolean>] [<then>])
<effect.add> ::= (add [<moves.piece>] <moves.to> [stack:boolean] [<then>])
<effect.deal> ::= (deal <dealableType> [<int>] [beginWith:<int>] [<then>])
<effect.remove> ::= (remove [<siteType>] (<int> |
               <sites>) [at:<whenType>] [<then>])
<effect.step> ::= (step [<moves.from>] [<direction>] <moves.to> [stack:boolean] [<then>])
<enclosed> ::= (enclosed [<siteType>] [<moves.from>] [<absoluteDirection>] [<apply>] [<then>])
<flip>     ::= (flip [<siteType>] [<int>] [<then>])
<fromTo>   ::= (fromTo <moves.from> <moves.to> [count:<int>] [copy:<boolean>] [stack:boolean] [<roleType>] [<
<hop>      ::= (hop [<moves.from>] [<direction>] [<moves.between>] <moves.to> [stack:boolean] [<then>])
<intervene> ::= (intervene [<moves.from>] [<absoluteDirection>] [<moves.between>] [<moves.to>] [<then>])
<leap>     ::= (leap [<moves.from>] {{<stepType>}} [forward:<boolean>] [rotations:<boolean>] <moves.to> [<the
<note>     ::= (note [<int> | <roleType>] string [to:<moves.player> |
               to:<roleType>])
<pass>     ::= (pass [<then>])
<playCard> ::= (playCard [<then>])
<promote>  ::= (promote [<siteType>] [<int>] <moves.piece> [<moves.player>
               | <roleType>] [<then>])
<propose>  ::= (propose (string | {string}) [<then>])
<push>     ::= (push [<moves.from>] <absoluteDirection> [<then>])
<roll>     ::= (roll [<then>])
<satisfy>  ::= (satisfy (<boolean> | {<boolean>}))
```

```
<select>   ::= (select <moves.from> [<moves.to>] [<then>])
<shoot>    ::= (shoot <moves.piece> [<moves.from>] [<absoluteDirection>] [<moves.to>] [<then>])
<slide>    ::= (slide [<moves.from>] [string] [<direction>] [<moves.between>] [<moves.to>] [<then>])
<sow>      ::= (sow [<siteType>] [<int>] [count:<int>] [string] [owner:<int>] [if:<boolean>] [apply:<nonDecis
<surround> ::= (surround [<moves.from>] [<relationType>] [<moves.between>] [<moves.to>] [except:<int>] [with:
<then>     ::= (then <nonDecision> [applyAfterAllMoves:boolean])
<trigger>  ::= (trigger string (<int> | <roleType>) [<then>])
<vote>     ::= (vote (string | {string}) [<then>])

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision

<nonDecision> ::= <effect> | <operator>

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect.requirement

<avoidStoredState> ::= (avoidStoredState <moves> [<then>])
<do>       ::= (do <moves> [next:<moves>] [ifAfterwards:<boolean>] [<then>])
<firstMoveOnTrack> ::= (firstMoveOnTrack [string] [<roleType>] <moves> [<then>])
<priority> ::= (priority <moves> <moves> [<then>]) |
               (priority {<moves>} [<then>])

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect.requirement.max

<max.max>  ::= (max Distance [string] [<roleType>] <moves> [<then>]) |
               (max <maxMovesType> <moves> [<then>])
<maxMovesType> ::= Captures | Moves

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect.set

<effect.set.set> ::= (set <setPlayerType> (<moves.player> |
               <roleType>) <int> [<then>]) |
               (set Pending [<int> | <sites>] [<then>]) |
               (set <setValueType> [<int>] [<then>]) |
               (set Direction [<moves.to>] [{<int>} |
               <int>] [previous:<boolean>] [next:<boolean>] [<then>])
               | (set NextPlayer (<moves.player> |
               <ints>) [<then>]) | (set TrumpSuit (<int> |
               <intArray.math.difference>) [<then>]) |
               (set <setSiteType> [<siteType>] at:<int> <int> [<then>])
               | (set <setRegionType> [<siteType>] (<int> |
               <sites>) [level:<int>] [<moves.player> |
               <roleType>] [stack:<boolean>] [<then>])
<setPlayerType> ::= Score | Value
<setRegionType> ::= Invisible | Masked | Visible
<setSiteType> ::= Count | State
<setValueType> ::= Counter | Pot | Var

//-----------------------------------------------------------------------------
```

```
// game.rules.play.moves.nonDecision.effect.state

<addScore> ::= (addScore ({<int>} |
               {<roleType>}) {<int>} [<then>]) |
               (addScore (<moves.player> | <roleType>) <int> [<then>])
<moveAgain> ::= (moveAgain)
<rememberState> ::= (rememberState [<then>])

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect.state.swap

<swap.swap> ::= (swap Players (<int> <roleType>) (<int> <roleType>) [<then>])
               | (swap Pieces [<int>] [<int>] [<then>])

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.effect.take

<take>     ::= (take Control (of:<roleType> |
               oF:<int>) (by:<roleType> |
               bY:<int>) [<siteType>] [<then>]) |
               (take Domino [<then>])

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.operators.foreach

<foreach.forEach> ::= (forEach Player [container:<int> |
               string] [<moves>] [<moves.player> |
               <roleType>] [top:<boolean>] [<siteType>] [<then>]) |
               (forEach Piece [string |
               {string}] [container:<int> |
               string] [<moves>] [<moves.player> |
               <roleType>] [top:<boolean>] [<siteType>] [<then>]) |
               (forEach Site <sites> <moves> [noMoveYet:<moves>] [<then>])
               |
               (forEach Direction [<moves.from>] [<direction>] [<moves.between>] <moves.to> [<then>])
               |
               (forEach Die [<int>] [combined:<boolean>] [replayDouble:<boolean>] [if:<boolean>] <moves> [<th

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.operator

<operator> ::= <allCombinations> | <logical.and> | <append> | <logical.if> |
               <logical.or>

//-----------------------------------------------------------------------------
// game.rules.play.moves.nonDecision.operators.logical

<allCombinations> ::= (allCombinations <moves> <moves> [<then>])
<append>   ::= (append <nonDecision> [<then>])
<logical.and> ::= (and <moves> <moves> [<then>]) | (and {<moves>} [<then>])
<logical.if> ::= (if <boolean> <moves> [<moves>] [<then>])
<logical.or> ::= (or {<moves>} [<then>]) | (or <moves> <moves> [<then>])
```

```
//-----------------------------------------------------------------------------
// game.rules.phase

<phase.phase> ::= (phase string [<roleType>] [<mode>] [<play>] [<end>] [<nextPhase>
              | {<nextPhase>}])
<nextPhase> ::= (nextPhase [<roleType> |
              <moves.player>] [<boolean>] [string])

//-----------------------------------------------------------------------------
// game.equipment.other

<dominoes> ::= (dominoes [upTo:int])
<hints>    ::= (hints [string] {<equipment.hint>} [<siteType>])
<map>      ::= (map [string] {int} {int}) | (map [string] {<math.pair>})
<regions>  ::= (regions [string] [<roleType>] ({int} | <sites> |
              {<sites>} | <regionTypeStatic> |
              {<regionTypeStatic>}) [string])

//-----------------------------------------------------------------------------
// game.equipment.container.board

<container.board.board> ::= (board <graph> [<board.track> |
              {<board.track>}] [use:<siteType>]) | <boardless> |
              <puzzleBoard> | <surakartaBoard>
<board.track> ::= (track string ({int} | string) [loop:boolean] [int |
              <roleType>] [directed:boolean])
<boardless> ::= (boardless <tilingBoardlessType>)

//-----------------------------------------------------------------------------
// game.equipment.container.board.custom

<surakartaBoard> ::= (surakartaBoard <graph> [loops:int] [from:int])

//-----------------------------------------------------------------------------
// game.equipment.container.board.puzzle

<puzzleBoard> ::= (puzzleBoard <graph> (<values> | {<values>}))

//-----------------------------------------------------------------------------
// game.mode

<mode>     ::= (mode [<modeType>])

//-----------------------------------------------------------------------------
// game.equipment

<equipment> ::= (equipment {<item>})
<item>     ::= <component> | <container> | <dominoes> | <hints> | <map> |
              <regions>

//-----------------------------------------------------------------------------
```

```
// game.equipment.component

<component> ::= <component.card> | <die> | <domino> | <component.piece> | <tile>
<component.card> ::= (card string <roleType> <cardType> rank:int value:int trumpRank:int trumpValue:int suit:
<component.piece> ::= (piece string [<roleType>] [<directionFacing>] [value:int] [<flips>] [<moves>])
<die>      ::= (die string <roleType> numFaces:int [<directionFacing>] [int] [<moves>])

//------------------------------------------------------------------------------
// game.equipment.component.tile

<tile>     ::= (tile string [<roleType>] [{<stepType>} |
               {{<stepType>}}] numSides:int [slots:{int} |
               slotsPerSide:int] [{<path>}] [<flips>] [<moves>])
<domino>   ::= (domino string <roleType> value:int value2:int [<moves>])
<path>     ::= (path from:int [slotsFrom:int] to:int [slotsTo:int] colour:int)

//------------------------------------------------------------------------------
// game.equipment.container

<container> ::= <container.board.board> | <deck> | <dice> | <hand>

//------------------------------------------------------------------------------
// game.equipment.container.other

<deck>     ::= (deck [<roleType>] [cardsBySuit:int] [suits:int] [{<equipment.card>}])
<dice>     ::= (dice [d:int] [faces:{int} |
               from:int] [<roleType>] num:int [biased:{int}])
<hand>     ::= (hand <roleType> [size:int])

//------------------------------------------------------------------------------
// game.rules

<rules.rules> ::= (rules [<meta>] [<start>] [<play>] phases:{<phase.phase>} [<end>])
               | (rules [<meta>] [<start>] <play> <end>)

//------------------------------------------------------------------------------
// game.rules.meta

<meta>     ::= (meta ({<metaRule>} | <metaRule>))
<automove> ::= (automove [boolean])
<meta.swap> ::= (swap [boolean])
<metaRule> ::= <automove> | <noRepeat> | <meta.swap>
<noRepeat> ::= (noRepeat [<repetitionType>])

//------------------------------------------------------------------------------
// game.rules.start

<start>    ::= (start ({<startRule>} | <startRule>))
<start.deal> ::= (deal <dealableType> [int])
<startRule> ::= <start.deal> | <place> | <deductionPuzzle.set> |
               <start.set.set> | <split>
```

```
//-----------------------------------------------------------------------------
// game.rules.start.deductionPuzzle

<deductionPuzzle.set> ::= (set [<siteType>] {{int}})

//-----------------------------------------------------------------------------
// game.rules.start.place

<place>    ::= (place Random {string} [count:{<int>}] <int> [invisibleTo:{<roleType>}] [maskedTo:{<roleType>]
               |
               (place Random [<int>] {string} [count:int] [invisibleTo:{<roleType>}] [maskedTo:{<roleType>}]
               | (place Stack (string |
               items:{string}) [string] [<siteType>] [<int> |
               {<int>} | <sites> | coord:string |
               {string}] [count:int |
               counts:{int}] [state:int] [rotation:int] [invisibleTo:{<roleType>}] [maskedTo:{<roleType>}])
               |
               (place string [<siteType>] [{<int>}] [<sites>] [{string}] [counts:{int}] [state:int] [rotation
               |
               (place string [string] [<siteType>] [<int>] [coord:string] [count:int] [state:int] [rotation:i
               |
               (place Random {<math.count>} <int> [invisibleTo:{<roleType>}] [maskedTo:{<roleType>}] [<siteTy

//-----------------------------------------------------------------------------
// game.rules.start.set

<start.set.set> ::= (set Team <int> {<roleType>}) |
               (set <setStartPlayerType> [<roleType>] <int>) |
               (set <setStartSitesType> int [<siteType>] (at:int |
               to:<sites>)) |
               (set <roleType> [<siteType>] [{<int>}] [<sites>] [{string}] [invisibleTo:{<roleType>}])
               |
               (set <roleType> [<siteType>] [<int>] [coord:string] [invisibleTo:{<roleType>}] [maskedTo:{<rol
               | (set AllInvisible [<siteType>])
<setStartPlayerType> ::= Amount | Score
<setStartSitesType> ::= Cost | Count | Phase

//-----------------------------------------------------------------------------
// game.rules.start.split

<split>    ::= (split Deck)

//-----------------------------------------------------------------------------
// game.rules.play

<play>     ::= (play <moves>)

//-----------------------------------------------------------------------------
// game.rules.end

<end>      ::= (end (<endRule> | {<endRule>}))
<byScore>  ::= (byScore [{<end.score>}])
```

```
<end.forEach> ::= (forEach [<roleType> | Track] if:<boolean> <result>)
<end.if>   ::= (if <boolean> [<end.if> | {<end.if>}] [<result>])
<endRule>  ::= <end.forEach> | <end.if>
<result>   ::= (result <roleType> <resultType>) | <byScore>

//-----------------------------------------------------------------------------
// game.match

<match>    ::= (match string [<players>] <games> <end>)
<games>    ::= (games (<subgame> | {<subgame>}))
<subgame>  ::= (subgame string [string] [next:<int>] [result:<int>])

//-----------------------------------------------------------------------------
// game.functions.booleans.was

<was>      ::= (was Pass)

//-----------------------------------------------------------------------------
// game.functions.booleans.no

<booleans.no.no> ::= (no Moves <roleType>)

//-----------------------------------------------------------------------------
// game.functions.booleans.math

<!=>       ::= (!= <sites> <sites>) | (!= (<int> |
               <roleType>) (<int> | <roleType>))
<<>        ::= (< <int> <int>)
<<=>       ::= (<= <int> <int>)
<=>        ::= (= <sites> <sites>) |
               (= <int> (<int> | <roleType>))
<>>        ::= (> <int> <int>)
<>=>       ::= (>= <int> <int>)
<booleans.math.if> ::= (if <boolean> <boolean> [<boolean>])
<math.and> ::= (and {<boolean>}) |
               (and <boolean> <boolean>)
<math.or>  ::= (or {<boolean>}) |
               (or <boolean> <boolean>)
<not>      ::= (not <boolean>)
<xor>      ::= (xor <boolean> <boolean>)

//-----------------------------------------------------------------------------
// game.functions.booleans.is

<booleans.is.is> ::= (is <isGraphType> <siteType>) |
               (is <isIndexPlayerType> [<siteType>] <int> (<int>
               | <roleType>)) | (is <isIntegerType> [<int>]) |
               (is <isComponentType> [<int>] [<siteType>] [at:<int>
               | in:<sites>] [<moves>]) |
               (is Related <relationType> [<siteType>] <int> (<int>
               | <sites>)) | (is RegularGraph (<moves.player> |
               <roleType>) [k:<int> | odd:<boolean> |
```

```
               even:<boolean>]) | (is <isPlayerType> (<int> |
               <roleType>)) | (is Triggered string (<int> |
               <roleType>)) | (is <isSimpleType>) |
               (is Crossing <int> <int>) |
               (is <isStringType> string) |
               (is Path <siteType> (<moves.player> |
               <roleType>) [length:<int> |
               maxLimit:<int>] [closed:boolean]) |
               (is <isSiteType> [<siteType>] <int>) |
               (is In [<int> | {<int>}] <sites>) |
               (is <isTreeType> (<moves.player> | <roleType>)) |
               (is Target [<int> | string] {int} [int | {int}]) |
               (is <isConnectType> [<siteType>] [int] [at:<int>] ({<sites>}
               | <roleType> | <regionTypeStatic>)) |
               (is Line [<siteType>] <int> [<absoluteDirection>] [through:<int>
               | throughAny:<sites>] [<roleType> |
               what:<int> |
               whats:{<int>}] [exact:<boolean>] [if:<boolean>] [byLevel:<boolean>])
               | (is Loop [<siteType>] [surround:<roleType> |
               {<roleType>}] [<absoluteDirection>] [<int>] [<int>
               | <sites>] [path:boolean])
<isComponentType> ::= Threatened | Within
<isConnectType> ::= Blocked | Connected
<isGraphType> ::= LastFrom | LastTo
<isIndexPlayerType> ::= Invisible | Masked | Visible
<isIntegerType> ::= AnyDie | Even | Flat | Odd | PipsMatch | SidesMatch | Visited
<isPlayerType> ::= Active | Enemy | Friend | Mover | Next | Prev
<isSimpleType> ::= Cycle | Full | Pending
<isSiteType> ::= Empty | Occupied
<isStringType> ::= Decided | Proposed
<isTreeType> ::= CaterpillarTree | SpanningTree | Tree | TreeCentre

//-----------------------------------------------------------------------------
// game.functions.booleans.deductionPuzzle.is

<deductionPuzzle.is.is> ::= (is <isPuzzleRegionResultType> [<siteType>] [<sites>] [of:<int>] [string] <int>)
               | (is Unique [<siteType>]) | (is Solved)
<isPuzzleRegionResultType> ::= Count | Sum

//-----------------------------------------------------------------------------
// game.functions.booleans.deductionPuzzle

<forAll>   ::= (forAll <puzzleElementType> <boolean>)

//-----------------------------------------------------------------------------
// game.functions.booleans.deductionPuzzle.all

<deductionPuzzle.all.all> ::= (all Different [<siteType>] [<sites>] [except:<int>
               | excepts:{<int>}])

//-----------------------------------------------------------------------------
// game.functions.region.sites.lineOfSight
```

```
<lineOfSightType> ::= Empty | Farthest | Piece

//-----------------------------------------------------------------------------
// game.functions.region.sites

<sites>    ::= (sites [<siteType>] [<int>] {{<stepType>}} [rotations:<boolean>])
               | (sites {<int>}) | (sites <sitesMoveType> <moves>) |
               (sites <sitesIndexType> [<siteType>] [<int>]) |
               (sites Side [<siteType>] [<moves.player> | <roleType> |
               <compassDirection>]) |
               (sites Distance [<siteType>] [<relationType>] from:<int> <int>)
               | (sites Random [<sites>] [num:<int>]) |
               (sites Crossing at:<int> [<moves.player> |
               <roleType>]) |
               (sites Group <siteType> of:<siteType> at:<int> [<moves.player>
               | <roleType>]) | (sites <sitesEdgeType>) |
               (sites <sitesSimpleType> [<siteType>]) |
               (sites [<siteType>] {string}) |
               (sites Incident <siteType> of:<siteType> at:<int> [owner:<moves.player>
               | <roleType>]) |
               (sites Around [typeLoc:<siteType>] (<int> |
               <sites>) [<regionTypeDynamic>] [distance:<int>] [<absoluteDirection>] [if:<boolean>] [includeS
               | (sites Direction (from:<int> |
               from:<sites>) [<direction>] [included:<boolean>] [stop:<boolean>] [distance:<int>] [<siteType>
               |
               (sites LineOfSight [<lineOfSightType>] [<siteType>] [at:<int>] [<direction>])
               | (sites [<moves.player> |
               <roleType>] [<siteType>] [string]) |
               (sites <sitesPlayerType> [<siteType>] [<moves.player> |
               <roleType>] [<nonDecision>] [string]) |
               (sites Start <moves.piece>) |
               (sites Occupied (by:<moves.player> |
               by:<roleType>) [container:<int> |
               container:string] [component:<int> |
               component:string |
               components:{string}] [top:boolean] [<siteType>])
<sitesEdgeType> ::= Angled | Axial | Horizontal | Slash | Slosh | Vertical
<sitesIndexType> ::= Cell | Column | Edge | Empty | Phase | Row | State
<sitesMoveType> ::= From | To
<sitesPlayerType> ::= Hand | Invisible | Masked | Track | Visible | Winning
<sitesSimpleType> ::= Board | Bottom | Centre | ConcaveCorners | ConvexCorners |
               Corners | Hint | Inner | LastFrom | LastTo | Left | LineOfPlay |
               Major | Minor | Outer | Pending | Playable | Right | ToClear |
               Top

//-----------------------------------------------------------------------------
// game.functions.region.math

<expand>   ::= (expand [<int> | string] (<sites> |
               origin:<int>) [steps:<int>] [<absoluteDirection>] [<siteType>])
<intersection> ::= (intersection {<sites>}) |
```

```
               (intersection <sites> <sites>)
<math.union> ::= (union {<sites>}) |
               (union <sites> <sites>)
<region.math.difference> ::= (difference <sites> (<sites> |
               <int>))
<region.math.if> ::= (if <boolean> <sites> [<sites>])

//-------------------------------------------------------------------------------
// game.functions.region.filter

<filter.forEach> ::= (forEach <sites> if:<boolean>)

//-------------------------------------------------------------------------------
// game.functions.graph.operators

<clip>     ::= (clip <graph> <poly>)
<complete> ::= (complete <graph> [eachCell:boolean])
<dual>     ::= (dual <graph>)
<hole>     ::= (hole <graph> <poly>)
<intersect> ::= (intersect {<graph>}) |
               (intersect <graph> <graph>)
<keep>     ::= (keep <graph> <poly>)
<layers>   ::= (layers <dim> <graph>)
<makeFaces> ::= (makeFaces <graph>)
<merge>    ::= (merge {<graph>} [connect:boolean]) |
               (merge <graph> <graph> [connect:boolean])
<operators.add> ::= (add [<graph>] [vertices:{{<float>}}] [edges:{{{<float>}}}
               |
               edges:{{<dim>}}] [edgesCurved:{{{<float>}}}] [cells:{{{<float>}}}
               | cells:{{<dim>}}] [connect:boolean])
<operators.remove> ::= (remove <graph> <poly> [trimEdges:boolean]) |
               (remove <graph> [cells:{{{float}}} |
               cells:{<dim>}] [edges:{{{float}}} |
               edges:{{<dim>}}] [vertices:{{float}} |
               vertices:{<dim>}] [trimEdges:boolean])
<operators.union> ::= (union {<graph>} [connect:boolean]) |
               (union <graph> <graph> [connect:boolean])
<recoordinate> ::= (recoordinate [<siteType>] [<siteType>] [<siteType>] <graph>)
<renumber> ::= (renumber [<siteType>] [<siteType>] [<siteType>] <graph>)
<rotate>   ::= (rotate <float> <graph>)
<scale>    ::= (scale <float> [<float>] [<float>] <graph>)
<shift>    ::= (shift <float> <float> [<float>] <graph>)
<skew>     ::= (skew float <graph>)
<splitCrossings> ::= (splitCrossings <graph>)
<subdivide> ::= (subdivide <graph> [min:<dim>])
<trim>     ::= (trim <graph>)

//-------------------------------------------------------------------------------
// game.functions.graph.generators.shape

<shape>    ::= (shape [Star] <dim>) | <rectangle> | <wedge>
<circle>   ::= (circle {<dim>} [stagger:boolean])
```

```
<rectangle> ::= (rectangle <dim> [<dim>] [diagonals:<diagonalsType>])
<repeat>   ::= (repeat <dim> <dim> step:{{float}} (<poly> |
               {<poly>}))
<shapeStarType> ::= Star
<spiral>   ::= (spiral turns:<dim> sites:<dim> [clockwise:boolean])
<wedge>    ::= (wedge <dim> [<dim>])

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.tri

<tri>      ::= (tri (<poly> | {<dim>})) |
               (tri [<triShapeType>] <dim> [<dim>])
<triShapeType> ::= Diamond | Hexagon | Limping | NoShape | Prism | Rectangle |
               Square | Star | Triangle

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.tiling

<tiling>   ::= (tiling <tilingType> (<poly> | {<dim>})) |
               (tiling <tilingType> <dim> [<dim>])
<tilingType> ::= T31212 | T333333_33434 | T33336 | T33344 | T33434 | T3464 |
               T3636 | T4612 | T488

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.square

<square>   ::= (square (<poly> |
               {<dim>}) [diagonals:<diagonalsType>]) |
               (square [<squareShapeType>] <dim> [diagonals:<diagonalsType>
               | pyramidal:boolean])
<diagonalsType> ::= Alternating | Concentric | Implied | Radiating | Solid
<squareShapeType> ::= Diamond | Limping | NoShape | Rectangle | Square

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.quadhex

<quadhex>  ::= (quadhex <dim> [thirds:boolean])

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.morris

<morris>   ::= (morris <dim> [joinCorners:<boolean>])

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.hex

<hex>      ::= (hex (<poly> | {<dim>})) |
               (hex [<hexShapeType>] <dim> [<dim>])
<hexShapeType> ::= Diamond | Hexagon | Limping | NoShape | Prism | Rectangle |
               Square | Star | Triangle

//-----------------------------------------------------------------------------
```

```
// game.functions.graph.generators.basis.celtic

<celtic>   ::= (celtic (<poly> | {<dim>})) |
               (celtic <dim> [<dim>])

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis.brick

<brick>    ::= (brick [<brickShapeType>] <dim> [<dim>] [trim:boolean])
<brickShapeType> ::= Diamond | Limping | Prism | Rectangle | Spiral | Square

//-----------------------------------------------------------------------------
// game.functions.graph.generators.basis

<basis>    ::= <brick> | <celtic> | <circle> | <hex> | <morris> | <quadhex> |
               <spiral> | <square> | <tiling> | <tri>

//-----------------------------------------------------------------------------
// game.functions.floats

<float>    ::= float

//-----------------------------------------------------------------------------
// game.functions.ints.value

<value>    ::= (value Player (<int> | <roleType>)) |
               (value Piece of:<int>) | (value Pending)

//-----------------------------------------------------------------------------
// game.functions.ints.trackSite

<trackSite> ::= (trackSite Move [from:<int>] [<roleType> |
               <moves.player> | string] steps:<int>) |
               (trackSite EndSite [<moves.player> | <roleType>] [string])

//-----------------------------------------------------------------------------
// game.functions.ints.tile

<pathExtent> ::= (pathExtent [<int>] [<int> | <sites>])

//-----------------------------------------------------------------------------
// game.functions.ints.state

<state>    ::= (state [<siteType>] at:<int> [level:<int>])
<amount>   ::= (amount (<roleType> | <moves.player>))
<counter>  ::= (counter)
<mover>    ::= (mover)
<next>     ::= (next)
<pot>      ::= (pot)
<prev>     ::= (prev)
<state.score> ::= (score (<moves.player> | <roleType>))
<what>     ::= (what [<siteType>] at:<int> [level:<int>])
```

```
<who>      ::= (who [<siteType>] at:<int> [level:<int>])

//-----------------------------------------------------------------------------
// game.functions.ints.stacking

<topLevel> ::= (topLevel [<siteType>] at:<int>)

//-----------------------------------------------------------------------------
// game.functions.ints.size

<size>     ::= (size [Group] [<siteType>] [from:<int>] [<moves.player> |
               <roleType>] [<absoluteDirection>]) |
               (size [Territory] [<siteType>] (<roleType> |
               <moves.player>) [<absoluteDirection>]) |
               (size Stack [<siteType>] [in:<sites> |
               at:<int> | string])

//-----------------------------------------------------------------------------
// game.functions.ints.math

<%>        ::= (% <int> <int>)
<ints.math.*> ::= (* {<int>}) | (* <int> <int>)
<ints.math.+> ::= (+ {<int>}) | (+ <int> <int>)
<ints.math.-> ::= (- [<int>] <int>)
<ints.math./> ::= (/ <int> <int>)
<ints.math.^> ::= (^ <int> <int>)
<ints.math.abs> ::= (abs <int>)
<ints.math.if> ::= (if <boolean> <int> <int>)
<ints.math.max> ::= (max <int> <int>)
<ints.math.min> ::= (min <int> <int>)

//-----------------------------------------------------------------------------
// game.functions.ints.match

<matchScore> ::= (matchScore <roleType>)

//-----------------------------------------------------------------------------
// game.functions.ints.last

<last>     ::= (last <lastType> [afterConsequence:<boolean>])
<lastType> ::= From | To

//-----------------------------------------------------------------------------
// game.functions.ints.dice

<dice.face> ::= (face <int>)
<pips>     ::= (pips)

//-----------------------------------------------------------------------------
// game.functions.intArray.math

<intArray.math.difference> ::= (difference {<int>} ({<int>} |
```

```
               <int>))

//------------------------------------------------------------------------------
// game.functions.range.math

<exact>    ::= (exact <int>)
<range.math.max> ::= (max <int>)
<range.math.min> ::= (min <int>)

//------------------------------------------------------------------------------
// game.functions.range

<range>    ::= (range int int) | (range <int> <int>) | <exact> |
               <range.math.max> | <range.math.min>

//------------------------------------------------------------------------------
// game.functions.intArray.state

<rotations> ::= (rotations (<absoluteDirection> | {<absoluteDirection>}))

//------------------------------------------------------------------------------
// game.functions.ints.count

<count.count> ::= (count Liberties [<siteType>] [in:<sites> |
               at:<int>] [<absoluteDirection>]) |
               (count Groups [<siteType>] [<roleType> |
               of:<int>] [min:<int>] [<absoluteDirection>]) |
               (count <countComponentType> [<siteType>] [<roleType> |
               of:<int>] [string] [in:<sites>]) |
               (count [<countSiteType>] [<siteType>] [in:<sites> |
               at:<int> | string]) |
               (count <countSimpleType> [<siteType>]) |
               (count Steps [<siteType>] [<relationType>] [<effect.step>] <int> <int>)
<countComponentType> ::= Pieces | Pips
<countSimpleType> ::= Active | Cells | Columns | Edges | Moves | MovesThisTurn |
               Phases | Players | Rows | Trials | Turns | Vertices
<countSiteType> ::= Adjacent | Diagonal | Neighbours | Off | Orthogonal | Sites

//------------------------------------------------------------------------------
// game.functions.ints.context

<context.between> ::= (between)
<context.edge> ::= (edge) | (edge <int> <int>)
<context.from> ::= (from [at:<whenType>])
<context.hint> ::= (hint)
<context.to> ::= (to)
<context.track> ::= (track)
<level>    ::= (level)
<site>     ::= (site)
<var>      ::= (var [string])

//------------------------------------------------------------------------------
```

```
// game.functions.ints.connection

<groupProduct> ::= (groupProduct [<siteType>] (<roleType> | <moves.player>))

//-----------------------------------------------------------------------------
// game.functions.ints.card

<card.card> ::= (card <cardSiteType> at:<int> [level:<int>]) |
                (card TrumpSuit)
<cardSiteType> ::= Rank | Suit | TrumpRank | TrumpValue

//-----------------------------------------------------------------------------
// game.functions.ints.board

<ahead>     ::= (ahead [<siteType>] <int> [steps:<int>] [<relativeDirection>
                | <absoluteDirection>])
<board.phase> ::= (phase [<siteType>] of:<int>)
<centrePoint> ::= (centrePoint [<siteType>])
<column>    ::= (column [<siteType>] of:<int>)
<coord>     ::= (coord [<siteType>] string)
<cost>      ::= (cost [<siteType>] (at:<int> | in:<sites>))
<handSite> ::= (handSite (<int> | <roleType>) [<int>])
<id>        ::= (id [string] [<roleType>])
<layer>     ::= (layer of:<int> [<siteType>])
<mapEntry> ::= (mapEntry [string] (<int> | <roleType>))
<row>       ::= (row [<siteType>] of:<int>)
<where>     ::= (where <int> [<siteType>]) |
                (where string (<int> |
                <roleType>) [state:<int>] [<siteType>])

//-----------------------------------------------------------------------------
// game.functions.dim.math

<dim.math.*> ::= (* {<dim>}) | (* <dim> <dim>)
<dim.math.+> ::= (+ {<dim>}) | (+ <dim> <dim>)
<dim.math.-> ::= (- <dim> <dim>)
<dim.math./> ::= (/ <dim> <dim>)
<dim.math.^> ::= (^ <dim> <dim>)
<dim.math.abs> ::= (abs <dim>)
<dim.math.max> ::= (max <dim> <dim>)
<dim.math.min> ::= (min <dim> <dim>)

//-----------------------------------------------------------------------------
// game.functions.dim

<dim>       ::= <dim.math.*> | <dim.math.+> | <dim.math.-> | <dim.math./> |
                <dim.math.^> | <dim.math.abs> | int | <dim.math.max> |
                <dim.math.min>

//-----------------------------------------------------------------------------
// game.functions.ints
```

```
<ints>     ::= {<int>} | <intArray.math.difference> | <regions> | <rotations>
<int>      ::= <%> | <dim.math.*> | <ints.math.*> | <dim.math.+> |
               <ints.math.+> | <dim.math.-> | <ints.math.-> | <dim.math./> |
               <ints.math./> | End | Off | Undefined | <dim.math.^> |
               <ints.math.^> | <dim.math.abs> | <ints.math.abs> | <ahead> |
               <amount> | <context.between> | <card.card> | <centrePoint> |
               <column> | <coord> | <cost> | <count.count> | <counter> |
               <context.edge> | <dice.face> | <context.from> | <groupProduct> |
               <handSite> | <context.hint> | <id> | <ints.math.if> | int |
               <last> | <layer> | <level> | <mapEntry> | <matchScore> |
               <dim.math.max> | <ints.math.max> | <dim.math.min> |
               <ints.math.min> | <mover> | <next> | <nextPhase> |
               <pathExtent> | <board.phase> | <pips> | <pot> | <prev> | <row> |
               <state.score> | <site> | <size> | <state> | <context.to> |
               <topLevel> | <context.track> | <trackSite> | <value> | <var> |
               <what> | <where> | <who>

//-----------------------------------------------------------------------------
// game.functions.booleans.can

<can>      ::= (can Move <moves>)

//-----------------------------------------------------------------------------
// game.functions.booleans.all

<booleans.all.all> ::= (all <allType>)
<allType>  ::= DiceEqual | DiceUsed | Passed

//-----------------------------------------------------------------------------
// game.functions.booleans

<boolean>  ::= <!=> | <<> | <<=> | <=> | <>> | <>=> | <booleans.all.all> |
               <deductionPuzzle.all.all> | <math.and> | boolean | <can> |
               <forAll> | <booleans.math.if> | <deductionPuzzle.is.is> |
               <booleans.is.is> | <booleans.no.no> | <not> | <math.or> |
               <was> | <xor>

//-----------------------------------------------------------------------------
// game.functions.directions

<directions> ::= (directions [<relativeDirection> |
               {<relativeDirection>}] [of:<relationType>] [bySite:boolean]) |
               (directions (<absoluteDirection> | {<absoluteDirection>}))
<directions.if> ::= (if <boolean> <direction> <direction>)

//-----------------------------------------------------------------------------
// game.util.end

<end.score> ::= (score <roleType> <int>)

//-----------------------------------------------------------------------------
// game.util.math
```

```
<math.count> ::= (count string <int>)
<math.pair> ::= (pair string string) | (pair <roleType> <roleType>) |
              (pair <roleType> int) | (pair int int) |
              (pair string <roleType>) | (pair <roleType> <landmarkType>) |
              (pair <roleType> string) | (pair int string)

//-----------------------------------------------------------------------------
// game.util.graph

<graph>    ::= (graph vertices:{{float}} [edges:{{int}}]) | <basis> | <brick> |
              <celtic> | <circle> | <clip> | <complete> | <dual> | <hex> |
              <hole> | <intersect> | <keep> | <layers> | <makeFaces> |
              <merge> | <morris> | <quadhex> | <recoordinate> |
              <operators.remove> | <renumber> | <repeat> | <rotate> |
              <scale> | <shape> | <shift> | <skew> | <spiral> |
              <splitCrossings> | <square> | <subdivide> | <tiling> | <tri> |
              <trim> | <operators.union> | <wedge>
<poly>     ::= (poly {{<dim>}}) | (poly {{float}})

//-----------------------------------------------------------------------------
// game.util.equipment

<equipment.card> ::= (card <cardType> rank:int value:int [trumpRank:int] [trumpValue:int] [biased:int])
<equipment.hint> ::= (hint int int) | (hint {int} [int])
<equipment.region> ::= <region.math.difference> | <expand> | <filter.forEach> |
              <region.math.if> | <intersection> | <sites> | <math.union>
<values>   ::= (values <siteType> <range>)

//-----------------------------------------------------------------------------
// game.util.moves

<flips>    ::= (flips int int)
<moves.between> ::= (between [before:<int>] [<range>] [after:<int>] [if:<boolean>] [trail:<int>] [<apply>])
<moves.from> ::= (from [<siteType>] [<sites> |
              <int>] [level:<int>] [if:<boolean>])
<moves.piece> ::= (piece (string | <int> | {string} |
              {<int>}) [state:<int>])
<moves.player> ::= (player <int>)
<moves.to> ::= (to [<siteType>] [<sites> |
              <int>] [level:<int>] [<rotations>] [if:<boolean>] [<apply>])

//-----------------------------------------------------------------------------
// game.util.directions

<direction> ::= <absoluteDirection> | <relativeDirection>
<absoluteDirection> ::= Adjacent | All | Angled | Axial | CCW | CW | D | DE |
              DN | DNE | DNW | DS | DSE | DSW | DW | Diagonal | Downward | E |
              ENE | ESE | In | N | NE | NNE | NNW | NW | Off | Orthogonal |
              Out | Rotational | S | SE | SSE | SSW | SW | SameLayer | U |
              UE | UN | UNE | UNW | US | USE | USW | UW | Upward | W | WNW |
              WSW
```

```
<compassDirection> ::= E | ENE | ESE | N | NE | NNE | NNW | NW | S | SE | SSE |
               SSW | SW | W | WNW | WSW
<directionFacing> ::= <absoluteDirection> | <directions> | <directions.if> |
               <relativeDirection>
<relativeDirection> ::= BL | BLL | BLLL | BR | BRR | BRRR | Backward |
               Backwards | FL | FLL | FLLL | FR | FRR | FRRR | Forward |
               Forwards | Leftward | Leftwards | OppositeDirection |
               Rightward | Rightwards | SameDirection

//-----------------------------------------------------------------------------
// game.types.component

<cardType> ::= Ace | Eight | Five | Four | Jack | Joker | King | Nine | Queen |
               Seven | Six | Ten | Three | Two
<dealableType> ::= Cards | Dominoes
<suitType> ::= Clubs | Diamonds | Hearts | Spades

//-----------------------------------------------------------------------------
// game.types.play

<modeType> ::= Alternating | Simulation | Simultaneous
<repetitionType> ::= InGame | InTurn | Infinite | Positional | Situational
<resultType> ::= Abandon | Crash | Draw | Loss | Tie | Win
<roleType> ::= All | Ally | Any | Each | Enemy | Mover | Neutral | Next |
               NonAlly | NonMover | NonNeutral | NonPartner | P1 | P10 | P11 |
               P12 | P13 | P14 | P15 | P16 | P2 | P3 | P4 | P5 | P6 | P7 | P8 |
               P9 | Partner | Player | Prev | Shared | Team1 | Team10 |
               Team11 | Team12 | Team13 | Team14 | Team15 | Team16 | Team2 |
               Team3 | Team4 | Team5 | Team6 | Team7 | Team8 | Team9
<whenType> ::= EndOfGame | EndOfMatch | EndOfMove | EndOfPhase | EndOfRound |
               EndOfSession | EndOfTurn | StartOfGame | StartOfMatch |
               StartOfMove | StartOfPhase | StartOfRound | StartOfSession |
               StartOfTurn

//-----------------------------------------------------------------------------
// game.types.board

<basisType> ::= Brick | Celtic | Circle | Dual | Hexagonal |
               HexagonalPyramidal | Mesh | Morris | NoBasis | QuadHex |
               Spiral | Square | SquarePyramidal | T31212 | T333333_33434 |
               T33336 | T33344 | T33434 | T3464 | T3636 | T4612 | T488 |
               Triangular
<landmarkType> ::= BottomSite | CentreSite | FirstSite | LastSite | LeftSite |
               RightSite | Topsite
<puzzleElementType> ::= Cell | Edge | Hint | Vertex
<regionTypeDynamic> ::= AllPlayers | Empty | Enemy | NotEmpty | NotEnemy |
               NotOwn | Own
<regionTypeStatic> ::= AllDirections | AllSites | Columns | Corners |
               Diagonals | HintRegions | Layers | Regions | Rows | Sides |
               SidesNoCorners | SubGrids | Touching | Vertices
<relationType> ::= Adjacent | All | Diagonal | Off | Orthogonal
<shapeType> ::= Circle | Cross | Custom | Diamond | Hexagon | Limping |
```

```
               NoShape | Polygon | Prism | Quadrilateral | Rectangle |
               Rhombus | Spiral | Square | Star | Triangle | Wedge | Wheel
<siteType> ::= Cell | Edge | Vertex
<stepType> ::= B | F | L | R
<tilingBoardlessType> ::= Hexagonal | Square | Triangular
```