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 1.1.19
May 17, 2021
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 ................................................. 18
  1.1 Strings ................................................. 18
  1.2 Booleans ............................................... 19
  1.3 Integers ............................................... 19
  1.4 Floats ................................................ 19

I Ludemes .................................................. 20

2 Game ...................................................... 21
  2.1 Game .................................................. 22
    2.1.1 game ............................................ 22
  2.2 Functions - TrackStep ................................ 23
    2.2.1 trackStep ....................................... 23
  2.3 Match ................................................ 24
    2.3.1 games .......................................... 24
    2.3.2 match .......................................... 24
    2.3.3 subgame ...................................... 25
  2.4 Mode ................................................ 27
    2.4.1 mode .......................................... 27
  2.5 Players .............................................. 28
    2.5.1 player ......................................... 28
    2.5.2 players ....................................... 28

3 Equipment ................................................. 30
  3.1 Equipment ........................................... 31
    3.1.1 equipment ..................................... 31
  3.2 Component ........................................... 32
    3.2.1 card ........................................... 32
    3.2.2 component .................................... 33
    3.2.3 die ............................................. 33
    3.2.4 piece .......................................... 33
  3.3 Component - Tile .................................... 35
    3.3.1 domino ......................................... 35
    3.3.2 path ........................................... 35
    3.3.3 tile ............................................ 36
3.4 Container - Board

3.4.1 board

3.4.2 boardless

3.4.3 track

3.5 Container - Board - Custom

3.5.1 mancalaBoard

3.5.2 surakartaBoard

3.6 Container - Other

3.6.1 deck

3.6.2 dice

3.6.3 hand

3.7 Other

3.7.1 dominoes

3.7.2 hints

3.7.3 map

3.7.4 regions

4 Graph Functions

4.1 Generators - Basis - Brick

4.1.1 brick

4.1.2 brickShapeType

4.2 Generators - Basis - Celtic

4.2.1 celtic

4.3 Generators - Basis - Hex

4.3.1 hex

4.3.2 hexShapeType

4.4 Generators - Basis - Quadhex

4.4.1 quadhex

4.5 Generators - Basis - Square

4.5.1 diagonalsType

4.5.2 square

4.5.3 squareShapeType

4.6 Generators - Basis - Tiling

4.6.1 tiling

4.6.2 tilingType

4.7 Generators - Basis - Tiling - Tiling3464

4.7.1 tiling3464ShapeType

4.8 Generators - Basis - Tri

4.8.1 tri

4.8.2 triShapeType

4.9 Generators - Shape

4.9.1 rectangle

4.9.2 regular

4.9.3 repeat

4.9.4 spiral

4.9.5 wedge

4.10 Generators - Shape - Concentric

4.10.1 concentric
### Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.10.2</td>
<td>66</td>
</tr>
<tr>
<td>4.11</td>
<td>67</td>
</tr>
<tr>
<td>4.11.1</td>
<td>67</td>
</tr>
<tr>
<td>4.11.2</td>
<td>68</td>
</tr>
<tr>
<td>4.11.3</td>
<td>68</td>
</tr>
<tr>
<td>4.11.4</td>
<td>69</td>
</tr>
<tr>
<td>4.11.5</td>
<td>69</td>
</tr>
<tr>
<td>4.11.6</td>
<td>70</td>
</tr>
<tr>
<td>4.11.7</td>
<td>71</td>
</tr>
<tr>
<td>4.11.8</td>
<td>71</td>
</tr>
<tr>
<td>4.11.9</td>
<td>72</td>
</tr>
<tr>
<td>4.11.10</td>
<td>72</td>
</tr>
<tr>
<td>4.11.11</td>
<td>73</td>
</tr>
<tr>
<td>4.11.12</td>
<td>73</td>
</tr>
<tr>
<td>4.11.13</td>
<td>75</td>
</tr>
<tr>
<td>4.11.14</td>
<td>75</td>
</tr>
<tr>
<td>4.11.15</td>
<td>76</td>
</tr>
<tr>
<td>4.11.16</td>
<td>76</td>
</tr>
<tr>
<td>4.11.17</td>
<td>77</td>
</tr>
<tr>
<td>4.11.18</td>
<td>77</td>
</tr>
<tr>
<td>4.11.19</td>
<td>78</td>
</tr>
<tr>
<td>4.11.20</td>
<td>78</td>
</tr>
<tr>
<td>4.11.21</td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td>81</td>
</tr>
<tr>
<td>5.1</td>
<td>82</td>
</tr>
<tr>
<td>5.1.1</td>
<td>82</td>
</tr>
<tr>
<td>5.1.2</td>
<td>82</td>
</tr>
<tr>
<td>5.1.3</td>
<td>83</td>
</tr>
<tr>
<td>5.1.4</td>
<td>83</td>
</tr>
<tr>
<td>5.1.5</td>
<td>84</td>
</tr>
<tr>
<td>5.1.6</td>
<td>84</td>
</tr>
<tr>
<td>5.1.7</td>
<td>85</td>
</tr>
<tr>
<td>5.1.8</td>
<td>85</td>
</tr>
<tr>
<td>6</td>
<td>86</td>
</tr>
<tr>
<td>6.1</td>
<td>87</td>
</tr>
<tr>
<td>6.1.1</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>88</td>
</tr>
<tr>
<td>6.2.1</td>
<td>88</td>
</tr>
<tr>
<td>6.2.2</td>
<td>88</td>
</tr>
<tr>
<td>6.2.3</td>
<td>89</td>
</tr>
<tr>
<td>6.2.4</td>
<td>89</td>
</tr>
<tr>
<td>6.2.5</td>
<td>90</td>
</tr>
<tr>
<td>6.2.6</td>
<td>90</td>
</tr>
<tr>
<td>6.2.7</td>
<td>90</td>
</tr>
<tr>
<td>6.2.8</td>
<td>91</td>
</tr>
<tr>
<td>6.2.9</td>
<td>91</td>
</tr>
</tbody>
</table>
6.2.10 * (mul) .................................................. 92
6.2.11 ∧ (pow) ................................................. 93
6.2.12 sin ......................................................... 93
6.2.13 sqrt ....................................................... 94
6.2.14 − (sub) ................................................... 94
6.2.15 tan ......................................................... 94

7 Rules ......................................................... 96
7.1 Rules ....................................................... 97
7.1.1 rules ..................................................... 97
7.2 End ......................................................... 99
7.2.1 byScore ................................................ 99
7.2.2 end ...................................................... 99
7.2.3 forEach ............................................... 99
7.2.4 if ......................................................... 100
7.2.5 payoffs ............................................... 100
7.2.6 result .................................................. 101
7.3 Meta ....................................................... 102
7.3.1 automove ............................................. 102
7.3.2 gravity ................................................. 102
7.3.3 meta ................................................... 102
7.3.4 noRepeat ............................................... 103
7.3.5 passEnd ............................................... 103
7.3.6 pin ..................................................... 104
7.3.7 swap .................................................. 104
7.4 Phase ...................................................... 105
7.4.1 nextPhase ............................................ 105
7.4.2 phase ................................................ 105
7.5 Play ....................................................... 107
7.5.1 play .................................................. 107
7.6 Start ....................................................... 108
7.6.1 deal .................................................. 108
7.6.2 start ................................................ 108
7.7 Start - DeductionPuzzle ................................. 110
7.7.1 set .................................................. 110
7.8 Start - ForEach .......................................... 111
7.8.1 forEach ............................................. 111
7.9 Start - Place ............................................. 113
7.9.1 place ............................................... 113
7.10 Start - Set .............................................. 117
7.10.1 set ................................................ 117
7.10.2 setStartPlayerType ................................. 119
7.10.3 setStartSitesType ................................. 119
7.11 Start - Split ............................................ 121
7.11.1 split .............................................. 121
8 Moves

8.1 Decision

8.1.1 move
8.1.2 moveMessageType
8.1.3 moveSimpleType
8.1.4 moveSiteType

8.2 NonDecision - Effect

8.2.1 add
8.2.2 apply
8.2.3 attract
8.2.4 bet
8.2.5 claim
8.2.6 custodial
8.2.7 deal
8.2.8 directional
8.2.9 enclose
8.2.10 flip
8.2.11 fromTo
8.2.12 hop
8.2.13 intervene
8.2.14 leap
8.2.15 note
8.2.16 pass
8.2.17 playCard
8.2.18 promote
8.2.19 propose
8.2.20 push
8.2.21 random
8.2.22 remove
8.2.23 roll
8.2.24 satisfy
8.2.25 select
8.2.26 shoot
8.2.27 slide
8.2.28 sow
8.2.29 step
8.2.30 surround
8.2.31 then
8.2.32 trigger
8.2.33 vote

8.3 NonDecision - Effect - Requirement

8.3.1 avoidStoredState
8.3.2 do
8.3.3 firstMoveOnTrack
8.3.4 priority
8.3.5 while

8.4 NonDecision - Effect - Requirement - Max

8.4.1 max
11 Integer Array Functions
11.1 Array ................................. 255
  11.1.1 array ................................ 255
11.2 Iterator .................................. 256
  11.2.1 team .................................. 256
11.3 Math .................................... 257
  11.3.1 difference ............................... 257
  11.3.2 intersection ............................ 257
  11.3.3 results ................................. 258
  11.3.4 union .................................. 258
11.4 Players .................................. 260
  11.4.1 players ................................ 260
  11.4.2 playersManyType ....................... 260
  11.4.3 playersTeamType ....................... 261
11.5 Sizes ................................... 262
  11.5.1 sizes .................................. 262
11.6 State ................................... 263
  11.6.1 rotations ............................... 263
11.7 Values .................................. 264
  11.7.1 values ................................ 264

12 Region Functions
12.1 Foreach ................................ 266
  12.1.1 forEach ................................ 266
12.2 Last .................................... 267
  12.2.1 last .................................. 267
12.3 Math .................................... 268
  12.3.1 difference ............................... 268
  12.3.2 expand ................................ 268
  12.3.3 if ..................................... 269
  12.3.4 intersection ............................ 269
  12.3.5 union .................................. 270
12.4 Sites ................................... 271
  12.4.1 lineOfSightType ....................... 271
  12.4.2 sites .................................. 271
  12.4.3 sitesEdgeType ......................... 281
  12.4.4 sitesIndexType ......................... 281
  12.4.5 sitesMoveType ......................... 281
  12.4.6 sitesPlayerType ....................... 281
  12.4.7 sitesSimpleType ....................... 282
16 Types

16.1 Board
   16.1.1 basisType
   16.1.2 hiddenData
   16.1.3 landmarkType
   16.1.4 puzzleElementType
   16.1.5 regionTypeDynamic
   16.1.6 regionTypeStatic
   16.1.7 relationType
   16.1.8 shapeType
   16.1.9 siteType
   16.1.10 stepType
   16.1.11 storeType
   16.1.12 tilingBoardlessType
   16.1.13 trackStepType

16.2 Component
   16.2.1 cardType
   16.2.2 dealableType
   16.2.3 suitType

16.3 Play
   16.3.1 modeType
   16.3.2 passEndTime
   16.3.3 prevType
   16.3.4 repetitionType
   16.3.5 resultType
   16.3.6 roleType
   16.3.7 whenType

II Metadata

17 Info Metadata

17.1 Metadata
   17.1.1 metadata

17.2 Info
   17.2.1 info

17.3 Info - Database
   17.3.1 aliases
   17.3.2 author
   17.3.3 classification
   17.3.4 credit
   17.3.5 date
   17.3.6 description
   17.3.7 origin
   17.3.8 publisher
   17.3.9 rules
   17.3.10 source
   17.3.11 version
19.5 Heuristics - Transformations ........................................ 376
19.5.1 divNumBoardSites ............................................. 376
19.5.2 divNumInitPlacement .......................................... 376
19.5.3 logisticFunction ................................................. 377
19.5.4 tanh .............................................................. 377
19.6 Misc ................................................................. 378
19.6.1 bestAgent ......................................................... 378
19.6.2 pair .............................................................. 378

III Metalanguage Features 380

20 Defines 381
20.1 Example .......................................................... 381
20.2 Parameters ........................................................ 382
20.3 Null Parameters .................................................. 382
20.4 Known Defines .................................................... 383

21 Options 384
21.1 Syntax ............................................................. 384
21.2 Option Priority ................................................... 385
21.3 Example .......................................................... 385

22 Rulesets 387
22.1 Example .......................................................... 387

23 Ranges 389
23.1 Smart Ranges ...................................................... 389

24 Constants 391
24.1 Off ................................................................. 391
24.2 End ................................................................. 391
24.3 Undefined ........................................................ 391

A Image List 394

B Known Defines 402
B.1 def/board .......................................................... 402
B.1.1 “AlquerqueBoard” ............................................. 402
B.1.2 “HoundsAndJackalsBoard” .................................. 403
B.1.3 “LascaBoard” .................................................... 403
B.2 def/conditions ...................................................... 404
B.2.1 “HandEmpty” .................................................... 404
B.2.2 “IsInCheck” ...................................................... 404
B.2.3 “NewTurn” ...................................................... 405
B.2.4 “NoPiece” ....................................................... 405
B.2.5 “NoSites” ....................................................... 405
B.2.6 “SameTurn” ..................................................... 406
B.3 def/functions ........................................................ 406
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 \((\text{game } \ldots)\) in game description files. Here, the dots (\ldots) 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
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.
- [mode]: The mode of the game [Alternating].
- <equipment>: The equipment of the game.
- <rules>: The rules of the game.

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)))
  )
)

)
2.2 Functions - TrackStep

Each track step can be a dim function (including positive integers), a compass direction or TrackStepType (Off/End/Repeat).

2.2.1 trackStep

Returns a track step entry.

Format

```
(trackStep (<dimFunction> | <compassDirection> | <trackStepType>))
```

where:

- `<dimFunction>`: Dim function or integer.
- `<compassDirection>`: Compass direction.
- `<trackStepType>`: Track step type: Off/End/Repeat.
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)
    }
   )
)
```

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 \(P_1, P_2, P_3, \ldots\) in game descriptions.

### 2.5.1 player

A player of the game.

**Format**

```
(player <directionFacing>)
```

where:

- `<directionFacing>`: The direction of the pieces of the player.

**Example**

```
(player N)
```

**Remarks**

Defines 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.

(\texttt{players \texttt{int}})

where:

• \texttt{int}: The number of players.

Examples

(\texttt{players \{ (player N) (player S) \}})

(\texttt{players 2})
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**

\[
\text{equipment } \{<\text{item}>\}
\]

where:

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

**Example**

\[
\text{equipment}
\{
  \begin{array}{l}
    \text{(board (square 3))} \\
    \text{(piece "Disc" P1)} \\
    \text{(piece "Cross" P2)}
  \end{array}
\}
\]

**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 \(<\text{item}>\) 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>] [maxState:int] [maxCount:int]
  [maxValue:int])

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.
- [maxState:int]: To set the maximum local state the game should check.
- [maxCount:int]: To set the maximum count the game should check.
- [maxValue:int]: To set the maximum value the game should check.

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>] [<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.
- `[<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] [flips] [moves] [maxState:int] [maxCount:int] [maxValue:int])
```

where:

- `<string>`: The name of the piece.
- `[roleType]`: The owner of the piece [Each].
- `[directionFacing]`: The direction 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.
- `[maxState:int]`: To set the maximum local state the game should check.
- `[maxCount:int]`: To set the maximum count the game should check.
- `[maxValue:int]`: To set the maximum value the game should check.

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:] [maxState:int] [maxCount:int] [maxValue:int])
```

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.
- **[maxState:int]**: To set the maximum local state the game should check.
- **[maxCount:int]**: To set the maximum count the game should check.
- **[maxValue:int]**: To set the maximum value the game should check.
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>}}]) ([<values>]
  ([{<values>}]) [use:<siteType>])
```

where:
- `<graphFunction>`: The graph function used to build the board.
- `[<track>]`: The track on the board.
- `[[<track>]]`: The tracks on the board.
- `<values>`: The range values of a graph element used for deduction puzzle.
- `[[<values>]]`: The range values of many graph elements used for deduction puzzle.
- `[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

The values range 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 element (i.e. vertex, edge and cell), each with a range of possible values.
3.4.2 boardless

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

Format

(boardless <tilingBoardlessType> [<dimFunction>])

where:

- <tilingBoardlessType>: The tiling of the boardless container.
- [<dimFunction>]: The "fake" size of the board used for boardless [41].

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> | {<trackStep>}) [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 cardinal.
- {<trackStep>}: Description using track steps. 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 mancalaBoard

Defines a Mancala-style board.

Format

```
(mancalaBoard int int [store:<storeType>] [numStores:int] ([<track>] | [{<track>}])))
```

where:
- `int`: The number of rows.
- `int`: The number of columns.
- `[store:<storeType>]`: The type of the store.
- `[numStores:int]`: The number of store.
- `<track>`: The track on the board.
- `[{<track>}]`: The tracks on the board.

Example

```
(mancalaBoard 2 6)
```

3.5.2 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]`.
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 - 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.

3.6.1 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)
  }
)

3.6.2 dice

Generates a set of dice.
Format

(dice [d:int] ([faces:{int}] | [facesByDie:{{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.
- [facesByDie:{{int}}]: The values of each face for each die.
- [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.

---

3.6.3 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.
3.7 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.

3.7.1 dominoes

Defines a dominoes set.

Format

(dominoes [upTo:int])

where:

• [upTo:int]: The number of dominoes [6].

Example

(dominoes)

3.7.2 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.

---

### 3.7.3 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.

3.7.4 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"}))
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 boards.
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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Prism</td>
<td>Prism board shape.</td>
</tr>
<tr>
<td>Spiral</td>
<td>Spiral board shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
</tbody>
</table>
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.

\[(\text{celtic} \ <\text{dimFunction}> \ [\text{<dimFunction>}]\)]

where:

- \(<\text{dimFunction}>\): Number of rows.
- \([\text{<dimFunction>}]\): Number of columns.

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

\[(\text{celtic} \ (<\text{poly}> \ | \ {\text{<dimFunction>}}))\]

where:

- \(<\text{poly}>\): Points defining the board shape.
- \(\{\text{<dimFunction>}\}\): Length of consecutive sides of outline shape.

Examples

\[(\text{celtic} \ 3)\]

\[(\text{celtic} \ (<\text{poly}> \ \{ \ {1 \ 2}\ \{1 \ 6\} \{3 \ 6\} \{3 \ 4\} \{4 \ 4\} \{4 \ 2\} \}))\]

\[(\text{celtic} \ \{4 \ 3 \ -1 \ 2\ 3\})\]

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.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoShape</td>
<td>No shape; custom graph.</td>
</tr>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Triangular board shape.</td>
</tr>
<tr>
<td>Hexagon</td>
<td>Hexagonal board shape.</td>
</tr>
<tr>
<td>Star</td>
<td>Multi-pointed star shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
<tr>
<td>Prism</td>
<td>Diamond shape extended vertically.</td>
</tr>
</tbody>
</table>
4.4 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.

4.4.1 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.

4.5 Generators - Basis - Square

This section contains the boards based on a square tiling.

4.5.1 diagonalsType

Defines how to handle diagonal relations on the Square tiling.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td>Diagonal connections (not edges) between opposite corners.</td>
</tr>
<tr>
<td>Solid</td>
<td>Solid edges between opposite diagonals, which split the square into four triangles.</td>
</tr>
<tr>
<td>Alternating</td>
<td>Every second diagonal is a solid edge, as per Alquerque boards.</td>
</tr>
<tr>
<td>Concentric</td>
<td>Concentric diagonal rings from the centre.</td>
</tr>
<tr>
<td>Radiating</td>
<td>Diagonals radiating from the centre.</td>
</tr>
</tbody>
</table>
4.5.2 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})
```

4.5.3 squareShapeType

Defines known shapes for the square tiling.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoShape</td>
<td>No shape; custom graph.</td>
</tr>
<tr>
<td>Shape</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
</tbody>
</table>
4.6 Generators - Basis - Tiling

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

4.6.1 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})

4.6.2 tilingType

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>


<table>
<thead>
<tr>
<th>Tiling</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T31212</td>
<td>Semi-regular tiling made up of triangles and dodecagons.</td>
</tr>
<tr>
<td>T3464</td>
<td>Rhombitrihexahedral tiling (e.g. Kensington).</td>
</tr>
<tr>
<td>T488</td>
<td>Semi-regular tiling made up of octagons with squares in the interstitial gaps.</td>
</tr>
<tr>
<td>T33434</td>
<td>Semi-regular tiling made up of squares and pairs of triangles.</td>
</tr>
<tr>
<td>T33336</td>
<td>Semi-regular tiling made up of triangles around hexagons.</td>
</tr>
<tr>
<td>T33344</td>
<td>Semi-regular tiling made up of alternating rows of squares and triangles.</td>
</tr>
<tr>
<td>T3636</td>
<td>Semi-regular tiling made up of triangles and hexagons.</td>
</tr>
<tr>
<td>T4612</td>
<td>Semi-regular tiling made up of squares, hexagons and dodecagons.</td>
</tr>
<tr>
<td>T333333.3434</td>
<td>Tiling 3.3.3.3.3.3.3.3.4.3.4.</td>
</tr>
</tbody>
</table>

### 4.7 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.

#### 4.7.1 tiling3464ShapeType

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Custom</td>
<td>Custom board shape.</td>
</tr>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Prism</td>
<td>Diamond board shape extended vertically.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Triangular board shape.</td>
</tr>
<tr>
<td>Hexagon</td>
<td>Hexagonal board shape.</td>
</tr>
<tr>
<td>Star</td>
<td>Multi-pointed star shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
</tbody>
</table>
4.8 Generators - Basis - Tri

This section contains the boards based on a triangular tiling.

4.8.1 tri

Defines a board on a triangular tiling.

Format

For defining a tri tiling with two dimensions.

\[(\text{tri } [<\text{triShapeType}>] \ <\text{dimFunction}> \ [\text{<dimFunction>}])\]

where:

- \[<\text{triShapeType}>]\: Board shape [Triangle].
- \[<\text{dimFunction}>]\: Board dimension; cells or vertices per side.
- \[<\text{dimFunction}>]\: Board dimension; cells or vertices per side.

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

\[(\text{tri } (<\text{poly} | \{<\text{dimFunction}>\}))\]

where:

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

Examples

\[(\text{tri } 8)\]
\[(\text{tri } \text{Hexagon } 3)\]
\[(\text{tri } (\text{poly } \{ \{ 1 \ 2 \} \ \{ 1 \ 6 \} \ \{ 3 \ 6 \} \ \{ 3 \ 4 \} )))\]
\[(\text{tri } \{ 4 \ 3 \ -1 \ 2 \ 3 \})\]

4.8.2 triShapeType

Defines known shapes for the triangular tiling.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoShape</td>
<td>No shape; custom graph.</td>
</tr>
<tr>
<td>Shape</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Triangular board shape.</td>
</tr>
<tr>
<td>Hexagon</td>
<td>Hexagonal board shape.</td>
</tr>
<tr>
<td>Star</td>
<td>Multi-pointed star shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
<tr>
<td>Prism</td>
<td>Diamond shape extended vertically.</td>
</tr>
</tbody>
</table>

4.9 Generators - Shape

This section contains different types of board shapes.

4.9.1 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)
```

4.9.2 regular

Defines a regular polygon.

Format

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

where:
- `<dimFunction>`: Number of sides.

Example

```
(regular 3)
```
4.9.3 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: {{-0.5 0.75} {1 0} }
  (poly {{0 0} {0 1} {1 1} {1 0}})
)
```

4.9.4 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].
4.9.5 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.
4.10 Generators - Shape - Concentric

This section contains boards based on tilings of concentric shapes.

4.10.1 concentric

Defines a board based on a tiling of concentric shapes.

Format

(concentric (<concentricShapeType> | sides:<dimFunction> | 
 {<dimFunction>} | [rings:<dimFunction>] [steps:<dimFunction>] 
 [midpoints:<boolean>] [joinMidpoints:<boolean>] 
 [joinCorners:<boolean>] [stagger:<boolean>])

where:
• <concentricShapeType>: Shape of board rings.
• sides:<dimFunction>: Number of sides (for polygonal shapes).
• {<dimFunction>}: Number of cells per circular ring.
• [rings:<dimFunction>]: Number of rings [3].
• [steps:<dimFunction>]: Number of steps for target boards with multiple sites per ring [null].
• [midpoints:<boolean>]: Whether to add vertices at edge midpoints [True].
• [joinMidpoints:<boolean>]: Whether to join concentric midpoints [True].
• [joinCorners:<boolean>]: Whether to join concentric corners [False].
• [stagger:<boolean>]: Whether to stagger cells in concentric circular rings [False].

Examples
(concentric Square rings:3)
(concentric Triangle rings:3 joinMidpoints:False joinCorners:True)
(concentric Hexagon rings:3 joinMidpoints:False joinCorners:True)
(concentric sides:5 rings:3)
(concentric {8})
(concentric {0 8})
(concentric {1 4 8} stagger:True)
(concentric Target rings:4)
(concentric Target rings:4 steps:16)

**Remarks**

Morris or Merels boards are examples of square concentric boards. Circular tilings are centred around a “pivot” point. For circular boards defined by a list of cell counts, the first count should be 0 (centre point) or 1 (centre cell).

**4.10.2 concentricShapeType**

Defines star shape types for known board types.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>Concentric squares rings, e.g. Morris boards.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Concentric triangles.</td>
</tr>
<tr>
<td>Hexagon</td>
<td>Concentric hexagons.</td>
</tr>
<tr>
<td>Target</td>
<td>Concentric circles, like a target.</td>
</tr>
</tbody>
</table>
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

\[
(add \ [\text{graphFunction}] \ [\text{vertices}]:\{\{\text{floatFunction}\}\}\]
\(\text{[(edges}:\{\{\text{floatFunction}\}\}\] | [edges}:\{\{\text{dimFunction}\}\]\)
\(\text{[edgesCurved}]:\{\{\text{floatFunction}\}\}\]) \text{[cells}]:\{\{\text{floatFunction}\}\}\]
\(\text{[connect}:\text{boolean}]\)
\]

where:

- \text{[graphFunction]}: The graph to remove elements from.
- \text{[vertices}]: Locations of vertices to add.
- \text{[edges}]: Locations of end points of edges to add.
- \text{[edgesCurved}]: Locations of end points and tangents of edges to add.
- \text{[cells}]: Locations of vertices of faces to add.
- \text{[connect}:\text{boolean}]\): Whether to connect newly added vertices to nearby neighbours [False].

Examples

\[(add \ (\text{square} \ 4) \ \text{vertices}:\{\ {1 \ 2}\ })\]
\[(add \ \text{edges}:\{\ {0 \ 0} \ {1 \ 1}\ }\])\]
\[(add \ (\text{square} \ 4) \ \text{cells}:\{\ {1 \ 1} \ {1 \ 2} \ {3 \ 2} \ {3 \ 1}\ })\]
\[(add \ (\text{square} \ 2) \ \text{edgesCurved}:\{\ {0 \ 0} \ {1 \ 0} \ {1 \ 2} \ {-1 \ 2}\ })\]

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
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 will be shown in isometric view from the side in a future version.

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.

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

where:

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

For removing some elements according to a polygon.

\[(\text{remove} \ <\text{graphFunction}> \ <\text{poly}> \ [\text{trimEdges}::\text{boolean}])]\]

where:

- \(<\text{graphFunction}>\): Graph to clip.
- \(<\text{poly}>\): Float points defining hole region.
- \([\text{trimEdges}::\text{boolean}]\): Whether to trim edges orphaned by removing faces [True].

Examples

\[(\text{remove} \ (\text{square} \ 4) \ \text{vertices}::\{\{0.0 \ 3.0\} \ {0.5 \ 2}\})\]

\[(\text{remove} \ (\text{square} \ 4) \ \text{cells}::0 \ 1 \ 2 \ \text{edges}::\{\{0 \ 1\} \ {1 \ 2\} \} \ \text{vertices}::\{1 \ 4\})\]

\[(\text{remove} \ (\text{square} \ 8) \ \text{poly} \ \{\ {2.5 \ 2.5\} \ {2.5 \ 5.5\} \ {4.5 \ 5.5\} \ {4.5 \ 4.5\} \ {5.5 \ 4.5\} \ {5.5 \ 2.5}\} \})\]
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

\[(\text{renumber} \ [\text{<siteType>}] \ [\text{<siteType>}] \ [\text{<siteType>}] \ <\text{graphFunction}>)]\]

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

Examples

\[(\text{renumber} \ (\text{merge} \ (\text{rectangle} \ 2 \ 5) \ (\text{square} \ 5)))\]
\[(\text{renumber} \ \text{Vertex} \ (\text{merge} \ (\text{rectangle} \ 2 \ 5) \ (\text{square} \ 5)))\]

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

\[(\text{rotate} \ <\text{floatFunction}> \ <\text{graphFunction}>)]\]

where:
- \[<\text{floatFunction}>\]: Number of degrees to rotate anticlockwise.
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))
```
(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.
Dim functions are ludemes that return a single integer value according to mathematical operations.
5.1 Math

Math functions return an integer value based on given inputs.

5.1.1 abs

Return the absolute value of a dim.

**Format**

\[
\text{(abs } \text{<dimFunction>})
\]

where:
- \text{<dimFunction>}: The value.

**Example**

\[
\text{(abs } (- 8 5))
\]

5.1.2 + (add)

Adds many values.

**Format**

To add two values.

\[
\text{(\{<dimFunction>\} <dimFunction>)}
\]

where:
- \text{<dimFunction>}: The first value.
- \text{<dimFunction>}: The second value.

To add all the values of a list.

\[
\text{(+ \{<dimFunction>\})}
\]

where:
- \text{<dimFunction>}: The list of the values.
Examples

(+ 5 2)
(+ {10 2 5})

5.1.3  / (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 and round down the result.

5.1.4  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)

5.1.5  \textbf{min}

Returns the minimum of two specified values.

\textbf{Format}

(min <dimFunction> <dimFunction>)

where:

- \(<dimFunction>\): The first value.
- \(<dimFunction>\): The second value.

Example

(min 20 3)

5.1.6  \textbf{* (mul)}

Returns to multiple of values.

\textbf{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})

5.1.7 ∧ (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)

5.1.8 − (sub)
Returns the subtraction A minus B.

Format

(− <dimFunction> <dimFunction>)
where:
• <dimFunction>: The value A.
• <dimFunction>: The value B.

Example

(− 5 1)
Float functions are ludemes that return a single floating point value produced by a specified function. They specify the amount of certain aspects of the game state, in a continuous rather than discrete space. Float parameters are always shown with their decimal point, e.g. 12.0, to clearly distinguish them from integer values.
6.1 ToFloat

6.1.1 toFloat
Converts a BooleanFunction or an IntFunction to a float.

Format

```plaintext
toFloat (<boolean> | <int>)
```

where:
- `<boolean>`: The boolean function.
- `<int>`: The int function.

Examples

```plaintext
toFloat (is Full))
toFloat (count Moves))
```
6.2 Math

Math functions return a float value based on given inputs.

6.2.1 abs

Return the absolute value of a float.

Format

(abs <floatFunction>)

where:

- <floatFunction>: The value.

Example

(abs (- 8.2 5.1))

6.2.2 + (add)

Adds many values.

Format

To add two values.
(+ <floatFunction> <floatFunction>)

where:

- <floatFunction>: The first value.
- <floatFunction>: The second value.

To add all the values of a list.
(+ {<floatFunction>})

where:

- {<floatFunction>}: The list of the values.
Examples

(+ 5.5 2.32)
(+ {10.1 2.8 5.1})

6.2.3 cos
Computes the cosine of a value.

Format

(cos <floatFunction>)
where:
  - <floatFunction>: The value.

Example

(cos 5.5)

6.2.4 / (div)
To divide a value by another.

Format

(/ <floatFunction> <floatFunction>)
where:
  - <floatFunction>: The first value.
  - <floatFunction>: The second value.

Example

(/ 5.5 2.32)
6.2.5 exp
Computes the exponential of a value.

Format

(exp <floatFunction>)
where:
  • <floatFunction>: The value.

Example

(exp 5.5)

6.2.6 log
Computes the logarithm of a value.

Format

(log <floatFunction>)
where:
  • <floatFunction>: The value.

Example

(log 5.5)

6.2.7 log10
Computes the logarithm 10 of a value.

Format

(log10 <floatFunction>)
where:
  • <floatFunction>: The value.
Example

\[(\log_{10} 5.5)\]

### 6.2.8 max

Returns the maximum of specified values.

**Format**

To get the maximum value between two.

\[(\text{max} \ <\text{floatFunction}> \ <\text{floatFunction}>\)]

where:
- \(<\text{floatFunction}>\): The first value.
- \(<\text{floatFunction}>\): The second value.

To get the maximum value in a list.

\[(\text{max} \ \{<\text{floatFunction}>\})\]

where:
- \(<\text{floatFunction}>\): The list of the values.

**Examples**

\[(\text{max} \ 5.5 \ 2.32)\]

\[(\text{max} \ \{10.1 \ 2.8 \ 5.1\})\]

### 6.2.9 min

Returns the minimum of specified values.

**Format**

To get the minimum value between two.

\[(\text{min} \ <\text{floatFunction}> \ <\text{floatFunction}>\)]

where:
• `<floatFunction>`: The first value.
• `<floatFunction>`: The second value.

To get the maximum value in a list.

\[
\text{(min } \{<\text{floatFunction}>\})
\]

where:
• `\{<\text{floatFunction}>\}`: The list of the values.

**Examples**

\[
\text{(min 5.5 2.32)}
\]

\[
\text{(min } \{10.1 2.8 5.1}\})
\]

---

**6.2.10  * (mul)**

Multiply many values.

**Format**

To multiply two values.

\[
(* \ <\text{floatFunction}> <\text{floatFunction}>)
\]

where:
• `<floatFunction>`: The first value.
• `<floatFunction>`: The second value.

To multiply all the values of a list.

\[
(* \ \{<\text{floatFunction}>\})
\]

where:
• `\{<\text{floatFunction}>\}`: The list of the values.
Examples

(* 5.5 2.32)
(* {10.1 2.8 5.1})

6.2.11  ∧ (pow)
Computes the first parameter to the power of the second parameter.

Format

(∧ <floatFunction> <floatFunction>)
where:
  • <floatFunction>: The first value.
  • <floatFunction>: The second value.

Example

(^ 5.5 2.32)

6.2.12  sin
Computes the sine of a value.

Format

(sin <floatFunction>)
where:
  • <floatFunction>: The value.

Example

(sin 5.5)
6.2.13  \texttt{sqrt}

Computes the square root of a value.

\textbf{Format}

\begin{verbatim}
(sqrt <floatFunction>)
\end{verbatim}

where:

- \texttt{<floatFunction>:} The first value.

\textbf{Example}

\begin{verbatim}
(sqrt 5.5)
\end{verbatim}

6.2.14  \texttt{\textbar} (\texttt{sub})

Returns the subtraction A minus B.

\textbf{Format}

\begin{verbatim}
(\textbar  <floatFunction> <floatFunction>)
\end{verbatim}

where:

- \texttt{<floatFunction>:} The value A.
- \texttt{<floatFunction>:} The value B.

\textbf{Example}

\begin{verbatim}
(\textbar  5.6 1.1)
\end{verbatim}

6.2.15  \texttt{tan}

Computes the tangent of a value.

\textbf{Format}

\begin{verbatim}
(tan <floatFunction>)
\end{verbatim}

where:

- \texttt{<floatFunction>:} The value.
Example

(tan 5.5)
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 game will typically have “start”, “play” and “end” rules.
7.1 Rules

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

7.1.1 rules

Sets the game’s rules.

**Format**

For defining the rules with start, play and end.

\[ \text{rules} \ [\text{<meta>}] \ [\text{<start>}] \ <\text{play}> \ <\text{end}> \]

where:
- \[\text{<meta>}\]: Metarules defined before play that supersede all other rules.
- \[\text{<start>}\]: Rules defining the starting position.
- \[\text{<play>}\]: Rules of play.
- \[\text{<end>}\]: Ending rules.

For defining the rules with some phases.

\[ \text{rules} \ [\text{<meta>}] \ [\text{<start>}] \ [\text{<play>}] \ \text{phases:}\{\text{<phase>}\} \ [\text{<end>}] \]

where:
- \[\text{<meta>}\]: Metarules defined before play that supersede all other rules.
- \[\text{<start>}\]: The starting rules.
- \[\text{<play>}\]: The playing rules shared between each phase.
- \[\text{phases:}\{\text{<phase>}\}\]: The phases of the game.
- \[\text{<end>}\]: 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)))
)
7.2 End

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

### 7.2.1 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)
```

### 7.2.2 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)))
```

### 7.2.3 forEach

Applies the end condition to each player of a certain type.
### 7.2.4 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.

### 7.2.5 payoffs

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

\[(\text{payoffs } \{<\text{payoff}>\})\]

where:
- \[<\text{payoff}>\]: The final score of each player.

Example

\[(\text{payoffs } \{ (\text{payoff P1 5.5}) (\text{payoff P2 1.2}) \})\]

7.2.6 result

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

Format

\[(\text{result } <\text{roleType}> <\text{resultType}>)\]

where:
- \[<\text{roleType}>\]: The player or the team.
- \[<\text{resultType}>\]: The result type of the player or team.

Example

\[(\text{result Mover Win})\]
7.3 Meta

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

7.3.1 automove

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

**Format**

```
(automove)
```

**Example**

```
(automove)
```

7.3.2 gravity

To apply a certain type of gravity after making a move.

**Format**

```
(gravity [PyramidalDrop])
```

**Example**

```
(gravity)
```

7.3.3 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))

---

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

Format

(noRepeat [\<repetitionType\>])

where:
- \[\<repetitionType\>\]: Type of repetition to forbid [Positional].

Example

(noRepeat PositionalInTurn)

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.

---

7.3.5 passEnd
To apply a certain end result to all players if all players pass their turns.

Format

(passEnd \<passEndType\>)

where:
- \<passEndType\>: The type of passEnd.
Example

(passEnd NoEnd)

---

### 7.3.6 pin

To apply a certain type of gravity after making a move.

**Format**

(pin SupportMultiple)

**Example**

(pin SupportMultiple)

---

### 7.3.7 swap

To activate the swap rule.

**Format**

(swap)

**Example**

(swap)
7.4 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.

7.4.1 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.

7.4.2 phase
Defines the phase of a game.

Format
(phrase <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

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

### Remarks

A phase can be defined for only one player.
7.5 Play

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

7.5.1 play

Checks the playing rules of the game.

**Format**

(play <moves>)

where:

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

**Example**

(play (forEach Piece))
7.6 Start

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

7.6.1 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)
```

7.6.2 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" "G5" "G6" "G7" "G8"\})
        (place "Knight1" \{"F3" "G4" "G9" "F10"\})
        (place "Pawn2" \{"K4" "K5" "K6" "K7" "K8" "J5" "J6" "J7" "J8"\})
        (place "Knight2" \{"K3" "J4" "J9" "K10"\})
    }
)

Remarks
For any game with starting rules, like pieces already placed on the board.
7.7 Start - DeductionPuzzle

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

7.7.1 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.
7.8 Start - ForEach

The `forEach` rules to initially run many starting rules in modifying a value parameter.

7.8.1 `forEach`

Iterates over a set of items.

**Format**

For iterating on teams.

```
(forEach Team <startRule>)
```

where:

- `<startRule>`: The starting rule to apply.

For iterating on values between two.

```
(forEach Site <region> [if:<boolean>] <startRule>)
```

where:

- `<region>`: The original region.
- `[if:<boolean>]`: The condition to satisfy.
- `<startRule>`: The starting rule to apply.

For iterating on values between two.

```
(forEach Value min:<int> max:<int> <startRule>)
```

where:

- `min:<int>`: The minimal value.
- `max:<int>`: The maximal value.
- `<startRule>`: The starting rule to apply.

For iterating through the players.

```
(forEach Player <startRule>)
```

where:

- `<startRule>`: The starting rule to apply.
For iterating through the players in using an IntArrayFunction.

\[(\text{forEach } <\text{IntArrayFunction}> <\text{startRule}>)\]

where:
- \(<\text{IntArrayFunction}>\): The list of players.
- \(<\text{startRule}>\): The starting rule to apply.

Examples

\[(\text{forEach Team (forEach (team) (set Hidden What at:1 to:Player)))}\]

\[(\text{forEach Site (sites Top) if:(is Even (site)) (place "Pawn1" (site)))}\]

\[(\text{forEach Value min:1 max:5 (set Hidden What at:10 level:(value) to:P1)})\]

\[(\text{forEach Player (set Hidden What at:1 to:Player))}\]

\[(\text{forEach (players Ally of:(next)) (set Hidden What at:1 to:Player))}\]
7.9 Start - Place

The place rules to initially place items into playing sites.

7.9.1 place

Sets some aspect of the initial game state.

Format

For placing an item to a site.

\[
\text{place } \langle\text{string}\rangle \ [\langle\text{string}\rangle] \ [\langle\text{siteType}\rangle] \ [\langle\text{int}\rangle] \ [\text{coord:<string>}] \\
[\text{count:<int>}] \ [\text{state:<int>}] \ [\text{rotation:<int>}] \ [\text{value:<int>}]\]
\]

where:

- *<string>*: The name of the item.
- *

For placing item(s) to sites.

\[
\text{place } \langle\text{string}\rangle \ [\langle\text{siteType}\rangle] \ {[\langle\text{int}\rangle]} \ {[\langle\text{region}\rangle]} \ {[\langle\text{string}\rangle]} \\
[\text{counts:{<int>}}] \ [\text{state:<int>}] \ [\text{rotation:<int>}] \ [\text{value:<int>}]\]
\]

where:

- *<string>*: The item to place.
- *


- **[value:<int>]**: The piece value to place [Undefined].

For placing items into a stack.

\[
\text{(place Stack (<string> | items:{<string>}) [<string>] [<siteType>]
\text{ ([[<int>] | [{<int>}] | [<region>] | [coord:<string>] |
\text{ [{<string>}]) ([count:<int>] ([counts:{<int>}]) [state:<int>]
\text{ [rotation:<int>] [value:<int>])}
\]

\text{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 [default SiteType of the board].
- **<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 [Undefined].
- **[rotation:<int>]**: The rotation value of the piece on the stack to place [Undefined].
- **[value:<int>]**: The piece value to place [Undefined].

For placing randomly pieces.

\[
\text{(place Random [<region>] {<string>} [count:<int>] [state:<int>]
\text{ [value:<int>] [<siteType>])}
\]

\text{where:}
- **<region>**: The region in which to randomly place piece(s).
- **{<string>}**: The names of the item to place.
- **[count:<int>]**: The number of items to place [1].
- **[state:<int>]**: The state value to place [Undefined].
- **[value:<int>]**: The piece value to place [Undefined].
- **[siteType]**: The graph element type [default SiteType of the board].
For placing randomly a stack.

```
(place Random {<string>} [count:{<int>}] [state:<int>] [value:<int>]
  <int> [<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.
- `[state:<int>]`: The state value to place [Undefined].
- `[value:<int>]`: The piece value to place [Undefined].
- `<int>`: The site on which to place the stack.
- `[<siteType>]`: The graph element type [default SiteType of the board].

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

```
(place Random {<count>} <int> [<siteType>])
```

where:
- `{<count>}`: The items to be placed, with counts.
- `<int>`: The site on which to place the stack.
- `[<siteType>]`: The graph element type [default SiteType of the board].

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))
)
7.10 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.

7.10.1 set

Sets some aspect of the initial game state.

Format

For setting the remembering values.

(set RememberValue [<string>] (<int> | <region>) [unique:<boolean>])

where:

- [<string>]: The name of the remembering values.
- <int>: The value to remember.
- <region>: The values to remember.
- [unique:<boolean>]: If True we remember a value only if not already remembered[False].

For setting the hidden information.

(set Hidden ([<hiddenData>] | [{<hiddenData>}] [<siteType>] (at:<int> | <region>) [level:<int>] [boolean] to:<roleType>)

where:

- [<hiddenData>]: The type of hidden data [Invisible].
- [{<hiddenData>}] : The types of hidden data [Invisible].
- [<siteType>]: The graph element type [default of the board].
- at:<int>: The site to set the hidden information.
- <region>: The region to set the hidden information.
- [level:<int>]: The level to set the hidden information [0].
- [boolean]: The value to set [True].
- to:<roleType>: The player with these hidden information.

For setting a site to a player.
(set <roleType> [<siteType>] [<int>] [coord:<string>])

where:
- <roleType>: The owner of the site.
- [<siteType>]: The graph element type [default SiteType of the board].
- [<int>]: The location to place a piece.
- [coord:<string>]: The coordinate of the location to place a piece.

For setting sites to a player.
(set <roleType> [<siteType>] [{<int>}] [<region>] [{<string>}])

where:
- <roleType>: The owner of the site.
- [<siteType>]: The graph element type [default SiteType of the board].
- [{<int>}]: The sites to fill.
- [<region>]: The region to fill.
- [{<string>}]: The coordinates of the sites to fill.

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 [default SiteType of the board].
- 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 index of the team.
  • \{<roleType>\}: The roleType of the player.

Examples

(set RememberValue 5)

(set Hidden What at:5 to:P1)

(set Hidden What at:6 to:P2)

(set Hidden Count (sites Occupied by:Next) to:P1)

(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\})

### 7.10.2 setStartPlayerType

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amount</td>
<td>Sets the initial amount for a player.</td>
</tr>
<tr>
<td>Score</td>
<td>Sets the initial score of a player.</td>
</tr>
</tbody>
</table>

### 7.10.3 setStartSitesType

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Sets the count of a site or region.</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>Cost</td>
<td>Sets the cost of a site or region.</td>
</tr>
<tr>
<td>Phase</td>
<td>Sets the phase of a site or region.</td>
</tr>
</tbody>
</table>
7.11 Start - Split

The (split ...) start ‘super’ ludeme to split objects between players.

7.11.1 split

Splits a deck of cards.

Format

(split Deck)

Example

(split Deck)

Remarks

This ludeme is used for card games.
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.
8.1 Decision

To specify that a move is a decision move.

8.1.1 move

Defines a decision move.

Format

For deciding to swap two players.

(move 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.

For deciding to swap two pieces.

(move 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 deciding to remove components.

(move Remove [siteType] (<int> | region) [level:int]

[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.
- [level:int]: The level to remove a piece [top level].
[at:<whenType>]: When to perform the removal [immediately].
[count:<int>]: The number of pieces to remove [1].
[<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 rotation.
(move Set Rotation [<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> [stack:<boolean>] [<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)))].
- [stack:<boolean>]: True if the move is applied to a stack [False].
- [<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 [default SiteType of the board].
- [<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 Swap Players P1 P2)

(move Swap Pieces (last To) (last From))

(move Remove (last To))

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

(move Set NextPlayer (player (mover)))

(move Set Rotation)

(move Set Rotation (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")
8.1.2 moveMessageType
Defines the types of decision move relative to a message.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propose</td>
<td>Makes a propose move.</td>
</tr>
<tr>
<td>Vote</td>
<td>Makes a vote move.</td>
</tr>
</tbody>
</table>

8.1.3 moveSimpleType
Defines the types of decision move corresponding to move with no parameters except the sub sequents.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pass</td>
<td>Makes a pass move.</td>
</tr>
<tr>
<td>PlayCard</td>
<td>Plays a card.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Makes a add move.</td>
</tr>
<tr>
<td>Claim</td>
<td>Makes a claim move.</td>
</tr>
</tbody>
</table>
8.2 NonDecision - Effect

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

8.2.1 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.

8.2.2 apply

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

Format

For checking a condition before to apply the default effect.

(apply if:<boolean>)

where:
• if:<boolean>: The condition to satisfy to get the effect moves.

For applying an effect.
(apply <nonDecision>)
where:
  • <nonDecision>: The moves to apply to make the effect.

For applying an effect if a condition is verified.
(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.

Examples

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

8.2.3 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 (last To))].
  • [<absoluteDirection>]: The specific direction [Adjacent].
  • [<then>]: The moves applied after that move is applied.
**Example**

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

### 8.2.4 **bet**

Is used to bet an amount.

**Format**

```plaintext
(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**

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

### 8.2.5 **claim**

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

**Format**

```plaintext
(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.

8.2.6 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 (last To))].
- [<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.
8.2.7 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))

8.2.8 directional
Is used to apply an effect to all the pieces in a direction from a location.

Format

(directional [<from>] [<direction>] [<to>] [<then>])

where:
- [<from>]: The origin of the move [(from (last To))].
- [<direction>]: The directions to use [(directions from:(last From) to:(last To))].
- [<to>]: The condition of the location to apply the effect [(to if:(is Enemy (to)) (apply (remove (from))))].
- [<then>]: The moves applied after that move is applied.

Example

(directional
  (from (last To))
  (to if:(is Enemy (who at:(to))) (apply (remove (to))))
  )
Remarks
For example, used in Fanorona.

8.2.9  enclose
Applies a move to an enclosed group.

Format

(enclose [<siteType>] [from] [direction] [between]
  [numException:<int>] [then])

where:
•  [<siteType>]: The graph element type [default site type of the board].
•  [from]: The origin of the enclosed group [(from (last To))].
•  [direction]: The direction to use [Adjacent].
•  [between]: The condition and effect on the pieces enclosed [(between if:(is Enemy (between)) (apply (remove (between))))].
•  [numException:<int>]: The number of liberties allowed in the group to enclose [0].
•  [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.

8.2.10  flip
Is used to flip a piece.
Format

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

where:
- [<siteType>]: The graph element type [default SiteType of the board].
- [<int>]: The location to flip the piece [(to)].
- [<then>]: The moves applied after that move is applied.

Examples

(flip)
(flip (last To))

8.2.11 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.

8.2.12 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.

8.2.13 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 (last To))].
  • [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))))
)

8.2.14 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.

8.2.15 note

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

Format

(note ([player:<int>] | [player:<roleType>]) (<string> (int> | <intArrayFunction> | <floatFunction> | <boolean> | <region> | <rangeFunction> | <direction> | <graphFunction>) ([to:<player>] (to:<roleType>)))

where:
• [player:<int>]: The index of the player to add at the beginning of the message.
• [player:<roleType>]: The role of the player to add at the beginning of the message.
• <string>: The message as a string.
• <int>: The message as an integer.
• <intArrayFunction>: The message as an array of integer.
• <floatFunction>: The message as a float.
• <boolean>: The message as a boolean.
• <region>: The message as a region.
• <rangeFunction>: The message as a range.
• <direction>: The message as a set of directions.
• `<graphFunction>`: The message as a graph.
• `[to:<player>]`: The index of the player.
• `[to:<roleType>]`: The role of the player [ALL].

Example

(note "Opponent has played")

8.2.16  pass

Passes this turn.

Format

(pass [<then>])

where:

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

Example

(pass)

8.2.17  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)

8.2.18 promote
Is used for promotion into another item.

Format

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

where:
- [<siteType>]: The graph element type [default SiteType of the board].
- [<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)

8.2.19 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")

---

8.2.20 push

Pushes all the pieces from a site in one direction.

Format

(push [<from>] <direction> [<then>])

where:
- [<from>]: Description of the “from” location [(from (last To))].
- <direction>: The direction to push.
- [<then>]: The moves applied after that move is applied.

Example

(push (from (last To)) E)

---

8.2.21 random

Returns a set of moves according to a set of probabilities.

Format

(random {<floatFunction>} {<moves>})

where:
- {<floatFunction>}: The different probabilities for each move.
- {<moves>}: The different possible moves.

Example

(random {0.01 0.99} { (forEach Piece) (move Pass) })

8.2.22 remove

Removes an item from a site.

Format

```
(remove [<siteType>] (<int> | <region>) [level:<int>] [at:<whenType>] [count:<int>] [<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.
- `[level:<int>]`: The level to remove a piece [top level].
- `[at:<whenType>]`: When to perform the removal [immediately].
- `[count:<int>]`: The number of pieces to remove [1].
- `[<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.

8.2.23 roll

Rolls the dice.

Format

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

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

(roll)

---

**8.2.24 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.

---

**8.2.25 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.

8.2.26 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.

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

Format

```
(slide [<from>] [<string>] [<direction>] [<between>] [<to>]
  [stack:<boolean>] [<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)))].
- [stack:<boolean>]: True if the move is applied to a stack [False].
- [<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.
8.2.28 sow

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

Format

\[
\text{sow } \left\langle \text{siteType} \right\rangle \left[ \left\langle \text{int} \right\rangle \left[ \text{count:<int>} \right] \left[ \text{string} \right] \left[ \text{owner:<int>} \right] \left[ \text{if:<boolean>} \right] \left[ \text{apply:<nonDecision>} \right] \left[ \text{includeSelf:<boolean>} \right] \left[ \text{origin:<boolean>} \right] \left[ \text{skipIf:<boolean>} \right] \left[ \text{backtracking:<boolean>} \right] \left[ \text{then} \right] \right\rangle
\]

where:
- \left\langle \text{siteType} \right\rangle: The graph element type [default SiteType of the board].
- \left\langle \text{int} \right\rangle: The origin of the sowing [(lastTo)].
- \text{count:<int>}: The number of pieces to sow [(count (lastTo))].
- \text{string}: The name of the track to sow [The first track if it exists].
- \text{owner:<int>}: The owner of the track.
- \text{if:<boolean>}: The condition to capture some counters after sowing [True].
- \text{apply:<nonDecision>}: The move to apply if the condition is satisfied.
- \text{includeSelf:<boolean>}: True if the origin is included in the sowing [True].
- \text{origin:<boolean>}: True to place a counter in the origin at the start of sowing [False].
- \text{skipIf:<boolean>}: The condition to skip a hole during the sowing.
- \text{backtracking:<boolean>}: Whether to apply the capture backwards from the “to” site.
- \text{then}: The moves applied after that move is applied.

Example

\[
\text{sow } (\text{if:(and} \left\langle \text{int} \right\rangle \text{is In (to) (sites Next))} \left\langle \text{int} \right\rangle \text{(or } (= \text{(count at:(to)) 2}) (= \text{(count at:(to)) 3})) \left\rangle \text{apply:(fromTo (from (to)) (to (mapEntry (mover))) count:(count at:(to))) includeSelf:False backtracking:True then})
\]
Remarks
Sowing is used in Mancala games. A track must be defined on the board in order for sowing to work.

8.2.29 step
Moves to a connected site.

Format

(\text{step } [\text{<from>}] [\text{<direction>}] \text{<to>} [\text{stack:<boolean>}] [\text{<then>}]\text{)}

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

Examples

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

Remarks
The (to ...) parameter is used to specify a condition on the site to step and on the effect to apply. But the region function in entry of that parameter is ignored because the region to move is implied by a step move.

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

(surround [from:] [relationType:] [between:] [to:] [except:] [with:] [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:] : The number of exceptions allowed to apply the effect [0].
• [with:] : 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))))
)

8.2.31 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.

8.2.32 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)

8.2.33 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")
8.3 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.

8.3.1 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.

8.3.2 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. The prior moves are not returned if that set of moves is empty.
- [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))
  )
  ifAfterwards:(is PipsMatch)
)

Remarks

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

8.3.3 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.

8.3.4 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.

8.3.5 while
Applies a move until the condition becomes false.

Format
(while <boolean> <moves> [<then>])

where:
- <boolean>: Conditions to make false thanks to the move to apply.
- <moves>: Moves to apply until the condition is false.
- [<then>]: The moves applied after that move is applied.

Example
(while (!= 100 (score P1)) (addScore P1 1))
8.4 NonDecision - Effect - Requirement - Max

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

8.4.1 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.

\[(\text{max} \ <\text{maxMovesType}> \ <\text{moves}> \ [\text{then}])\]

where:
- \(<\text{maxMovesType}>\): The type of property to maximise.
- \(<\text{moves}>\): The moves to filter.
- \([\text{then}]\): The moves applied after that move is applied.

For getting the moves with the max distance.

\[(\text{max} \ \text{Distance} \ [\text{string}] \ [\text{roleType}] \ <\text{moves}> \ [\text{then}])\]

where:
- \([\text{string}]\): The name of the track.
- \([\text{roleType}]\): The owner of the track.
- \(<\text{moves}>\): The moves to filter.
- \([\text{then}]\): The moves applied after that move is applied.

**Examples**

\[(\text{max} \ \text{Moves} \ (\text{forEach} \ \text{Piece}))\]
\[(\text{max} \ \text{Captures} \ (\text{forEach} \ \text{Piece}))\]
\[(\text{max} \ \text{Distance} \ (\text{forEach} \ \text{Piece}))\]

8.4.2 maxMovesType

Defines the types of properties which can be used for the Max super ludeme with only a move ludeme in entry.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Moves</td>
<td>To filter a list of legal moves to keep only the moves allowing the maximum number of moves in a turn.</td>
</tr>
<tr>
<td>Captures</td>
<td>To filter a list of moves to keep only the moves doing the maximum possible number of captures.</td>
</tr>
</tbody>
</table>
8.5 NonDecision - Effect - Set

The \(\text{set} \ldots\) ‘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.

8.5.1 set

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

Format

For setting a team.

\[
\text{set Team } \langle\text{int}\rangle\ \{\langle\text{roleType}\rangle\} \ [\langle\text{then}\rangle]
\]

where:

- \(\langle\text{int}\rangle\): The index of the team.
- \(\{\langle\text{roleType}\rangle\}\): The roleType of each player on the team.
- \([\langle\text{then}\rangle]\): The moves applied after that move is applied.

For setting the hidden information.

\[
\text{set Hidden } ([\langle\text{hiddenData}\rangle] \ | \ [[\langle\text{hiddenData}\rangle]]) \ [\langle\text{siteType}\rangle]
\]

\[
\quad \text{at:}\langle\text{int}\rangle \ | \ \langle\text{region}\rangle) \ [\langle\text{level:}\langle\text{int}\rangle\rangle] \ [\langle\text{boolean}\rangle] \ (\text{to:}\langle\text{player}\rangle \ | \ \langle\text{to:}\langle\text{roleType}\rangle\rangle) \ [\langle\text{then}\rangle]]
\]

where:

- \([\langle\text{hiddenData}\rangle]\): The type of hidden data [Invisible].
- \([\langle\text{hiddenData}\rangle]\): The types of hidden data [Invisible].
- \([\langle\text{siteType}\rangle]\): The graph element type [default of the board].
- \(\text{at:}\langle\text{int}\rangle\): The site to set the hidden information.
- \(\langle\text{region}\rangle\): The region to set the hidden information.
- \([\langle\text{level:}\langle\text{int}\rangle\rangle]\): The level to set the hidden information [0].
- \([\langle\text{boolean}\rangle]\): The value to set [True].
- \(\text{to:}\langle\text{player}\rangle\): The player with these hidden information.
- \(\text{to:}\langle\text{roleType}\rangle\): The roleType with these hidden information.
- \([\langle\text{then}\rangle]\): The moves applied after that move is applied.
For setting the trump suit.

\[(\text{set TrumpSuit} \ (<\text{int}> \ | \ <\text{difference}>) \ [\text{then}])\]

where:

- \(<\text{int}>\): The suit to choose.
- \(<\text{difference}>\): The possible suits to choose.
- \([\text{then}]\): The moves applied after that move is applied.

For setting the next player.

\[(\text{set NextPlayer} \ (<\text{player}> \ | \ <\text{IntArrayFunction}>)) \ [\text{then}])\]

where:

- \(<\text{player}>\): The data of the next player.
- \(<\text{IntArrayFunction}>\): The indices of the next players.
- \([\text{then}]\): The moves applied after that move is applied.

For setting the rotations.

\[(\text{set Rotation} \ [<\text{to}>] \ ([\{<\text{int}>\}] \ | \ [<\text{int}>]) \ [\text{previous:<boolean}>] \ [\text{next:<boolean}>] \ [\text{then}])\]

where:

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

For setting the value or the score of a player.

\[(\text{set <setPlayerType>} \ (<\text{player}> \ | \ <\text{roleType}>)) \ <\text{int}> \ [\text{then}]\]

where:

- \(<\text{setPlayerType}>\): The type of property to set.
- \(<\text{player}>\): The index of the player.
- \(<\text{roleType}>\): 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 Var [<string>] [<int>] [<then>])
```

where:

• `<string>`: The name of the var.
• `<int>`: The new counter value [-1].
• `[<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> [level:<int>] <int> [<then>])
```

where:

• `<setSiteType>`: The type of property to set.
• `[<siteType>]`: The graph element type [default SiteType of the board].
• `at:<int>`: The site to set.
• [level:<int>]: The level to set [0].
• <int>: The new value.
• [<then>]: The moves to apply afterwards.

Examples

(set Team 1 { P1 P3})
(set Hidden What at:(last To) to:Mover)
(set Hidden What at:(last To) to:P2)
(set Hidden Count (sites Occupied by:Next) to:Mover)
(set TrumpSuit (card Suit at:(handSite Shared)))
(set NextPlayer (player (mover)))
(set Rotation)
(set Rotation (to (last To)) next:False)
(set Value Mover 1)
(set Score P1 50)
(set Pending)
(set Pending (sites From (forEach Piece)))
(set Var (value Piece at:(last To)))
(set Counter -1)
(set State at:(last To) (mover))
(set Count at:(last To) 10)
(set Value at:(last To) 10)

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Sets the value associated with a player.</td>
</tr>
<tr>
<td>Score</td>
<td>Sets the score of a player.</td>
</tr>
</tbody>
</table>
8.5.3 setSiteType
Defines properties of sites that can be set in the game state.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>Set the count value for specified sites.</td>
</tr>
<tr>
<td>State</td>
<td>Set the local state value for specified sites.</td>
</tr>
<tr>
<td>Value</td>
<td>Set the piece value for specified sites.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>Sets the counter of the game state.</td>
</tr>
<tr>
<td>Pot</td>
<td>Sets the pot of the game state.</td>
</tr>
</tbody>
</table>
8.6 NonDecision - Effect - State

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

8.6.1 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 })
```

8.6.2 moveAgain

Is used to move again.
Format

(moveAgain [<then>])

where:

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

Example

(moveAgain)

Remarks

For games with multiple moves in a turn.
8.7 NonDecision - Effect - State - Forget

The \texttt{(forget ...)} ‘super’ ludeme forgets some information stored.

8.7.1 \texttt{forget}

Forget information about the state to be used in future state.

\textbf{Format}

For forgetting all values.

\texttt{(forget Value \{<string>\} All \{<then>\})}

where:

\begin{itemize}
\item \{<string>\}: The name of the remembering values.
\item \{<then>\}: The moves applied after that move is applied.
\end{itemize}

For forgetting a value.

\texttt{(forget Value \{<string>\} <int> \{<then>\})}

where:

\begin{itemize}
\item \{<string>\}: The name of the remembering values.
\item <int>: The value to forget.
\item \{<then>\}: The moves applied after that move is applied.
\end{itemize}

\textbf{Examples}

\texttt{(forget Value All)}

\texttt{(forget Value (count Pips))}
8.8 NonDecision - Effect - State - Remember

The (remember ...) ‘super’ ludeme remembers some information to use them in future states.

8.8.1 remember

Remember information about the state to be used in future state.

Format

For remembering a value.

(remember Value [<string>] <int> [unique:<boolean>] [<then>])

where:
- [<string>]: The name of the remembering values.
- <int>: The value to remember.
- [unique:<boolean>]: True if each remembered value has to be unique [False].
- [<then>]: The moves applied after that move is applied.

For remembering the current state.

(remember State [<then>])

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

Examples

(remember Value (count Pips))
(remember State)
8.9 NonDecision - Effect - State - Swap

The (swap ...) ‘super’ ludeme swaps two pieces or two players.

8.9.1 swap

Swaps two players or two pieces.

**Format**

For swapping two pieces.

\[
\text{(swap Pieces } \langle \text{int} \rangle \ [\langle \text{int} \rangle \ [\langle \text{then} \rangle)])
\]

where:

- \(\langle \text{int} \rangle\): The first location \([\text{lastFrom}]\).
- \(\langle \text{int} \rangle\): The second location \([\text{lastTo}]\).
- \(\langle \text{then} \rangle\): The moves applied after that move is applied.

For swapping two players.

\[
\text{(swap Players } \langle \text{int} \rangle \ \langle \text{roleType} \rangle \ \langle \text{int} \rangle \ \langle \text{roleType} \rangle \ [\langle \text{then} \rangle])
\]

where:

- \(\langle \text{int} \rangle\): The index of the first player.
- \(\langle \text{roleType} \rangle\): The role of the first player.
- \(\langle \text{int} \rangle\): The index of the second player.
- \(\langle \text{roleType} \rangle\): The role of the second player.
- \(\langle \text{then} \rangle\): The moves applied after that move is applied.

**Examples**

\[
\text{(swap Pieces)}
\]

\[
\text{(swap Players P1 P2)}
\]
8.10 NonDecision - Effect - Take

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

8.10.1 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>)
  ([at:<int>] ([to:<region>]) [<siteType>] [<then>]))

where:

- of:<roleType>: The roleType of the pieces to take control of.
- of:<int>: The player index of the pieces to take control of.
- by:<roleType>: The roleType taking the control.
- by:<int>: The player index of the player taking control.
- [at:<int>]: The site to take the control.
- [to:<region>]: The region to take the control.
- [<siteType>]: The graph element type [default SiteType of the board].
- [<then>]: The moves applied after that move is applied.

**Examples**

(take Domino)

(take Control of:P1 by:Mover)
8.11 NonDecision - Operators - Foreach

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

8.11.1 forEach

Iterates over a set of items.

**Format**

For iterating on teams.

```
(forEach Level [<siteType>] <int> <moves> [<then>])
```

where:

- `<siteType>`: The type of the graph elements of the group [default SiteType of the board].
- `<int>`: The site to iterate through.
- `<moves>`: The moves.
- `<then>`: The moves applied after that move is applied.

For iterating on teams.

```
(forEach Team <moves> [<then>])
```

where:

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

For iterating through the groups.

```
(forEach Group [<siteType>] [<direction>] [if:<boolean>] <moves> [<then>])
```

where:

- `<siteType>`: The type of the graph elements of the group.
- `<direction>`: The directions of the connection between elements in the group [Adjacent].
- `<if:boolean>`: The condition on the pieces to include in the group.
- `<moves>`: The moves to apply.
• \(<\text{then}>\): The moves applied after that move is applied.

For iterating through the dice.

(forEach Die \(<\text{int}>\) \[\text{combined:<boolean>}\] \[\text{replayDouble:<boolean>}\] 
\[\text{if:<boolean>}\] \(<\text{moves}>\) \[\text{then}>\])

where:
  • \(<\text{int}>\): The index of the dice container [0].
  • \[\text{combined:<boolean>}\]: True if the combination is allowed [False].
  • \[\text{replayDouble:<boolean>}\]: True if double allows a second move [False].
  • \[\text{if:<boolean>}\]: The condition to satisfy to move [True].
  • \(<\text{moves}>\): The moves to apply.
  • \[\text{then}>\]: The moves applied after that move is applied.

For iterating through the directions.

(forEach Direction \[\text{from}>\] \[\text{direction}>\] \[\text{between}>\] (<to> | \(<\text{moves}>\) \[\text{then}>\])

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

For iterating through the sites of a region.

(forEach Site \(<\text{region}>\) \(<\text{moves}>\) \[\text{noMoveYet:<moves>}\] \[\text{then}>\])

where:
  • \(<\text{region}>\): The region used.
  • \(<\text{moves}>\): The move to apply.
  • \[\text{noMoveYet:<moves>}\]: The moves to apply if the list of moves resulting from the generator is empty.
• \[<\text{then}>\]: The moves applied after that move is applied.

For iterating through values from an IntArrayFunction.

(forEach Value \(<\text{intArrayFunction}> <\text{moves}> [<\text{then}>]\))

where:
• \(<\text{intArrayFunction}>\): The IntArrayFunction.
• \(<\text{moves}>\): The move to apply.
• \([<\text{then}>]\): The moves applied after that move is applied.

For iterating through values between two.

(forEach Value min:<int> max:<int> <moves> [then])

where:
• \(\text{min:<int>}\): The minimal value.
• \(\text{max:<int>}\): The maximal value.
• \(<\text{moves}>\): The move to apply.
• \([<\text{then}>]\): The moves applied after that move is applied.

For iterating through the pieces.

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

where:
• \([on:<siteType>]\): Type of graph element.
• \(<\text{string}>\): The name of the piece.
• \([<\text{string}>]\): The names of the pieces.
• \([\text{container:<int>}\]): The index of the container.
• \(<\text{string}>\): The name of the container.
• \([<\text{moves}>]\): The specific moves to apply to the pieces.
• \([<\text{player}>]\): The owner of the piece \([\text{mover}]\).
• \([<\text{roleType}>]\): The role type of the owner of the piece \([\text{Mover}]\).
• \([\text{top:<boolean>}]\): To apply the move only to the top piece in case of a stack \([\text{False}]\).
• \([<\text{then}>]\): The moves applied after that move is applied.
For iterating through the players.

(forEach Player <moves> [<then>])

where:

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

For iterating through the players in using an IntArrayFunction.

(forEach <IntArrayFunction> <moves> [<then>])

where:

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

Examples
(forEach Level
   (last To)
   (if
      (= (id "King" Mover) (what at:(last To) level:(level)))
      (addScore Mover 1)
   )
)

(forEach Team (forEach (team) (set Hidden What at:(site) to:Player)))

(forEach Group (addScore Mover (count Sites in:(sites))))

(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 Value
   (values Remembered)
   (move (from) (to (trackSite Move steps:(value))))
)

(forEach Value min:1 max:5 (move (from) (to (trackSite Move steps:(value))))

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

(forEach Player (addScore (player (player)) 1))

(forEach (players Ally of:(next)) (addScore (player (player)) 1))
8.12 NonDecision - Operators - Logical

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

8.12.1 allCombinations

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

**Format**

\[
\text{(allCombinations } \text{<moves> } \text{<moves> } [\text{<then>}]\text{)}
\]

where:
- \text{<moves>}: The first list.
- \text{<moves>}: The second list.
- \text{[<then>]}: The moves applied after that move is applied.

**Example**

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

8.12.2 and

Is used to combine lists of moves.

**Format**

For making a move between two sets of moves.

\[
\text{(and } \text{<moves> } \text{<moves> } [\text{<then>}]\text{)}
\]

where:
- \text{<moves>}: The first move.
- \text{<moves>}: The second move.
- \text{[<then>]}: The moves applied after that move is applied.
For making a move between many sets of moves.

\[
\text{\texttt{(and}} \{\texttt{<moves>}}\} \texttt{[<then>\}])
\]

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

### Examples

\[
\text{\texttt{(and (set Score P1 100) (set Score P2 100))}}
\]

\[
\text{\texttt{(and \{ (set Score P1 100) (set Score P2 100) (set Score P3 100) \}])}}
\]

### 8.12.3 append

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

### Format

\[
\text{\texttt{(append \texttt{<nonDecision>}} \texttt{[<then>\]})}
\]

where:
- \texttt{\textless nonDecision\textgreater}: The moves to merge.
- \texttt{\textless then\textgreater}: 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)))
    )
  )
)

8.12.4 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 satisfied.
• [<then>]: The moves applied after that move is applied.

Examples

(if (is Mover P1) (moveAgain))

(if (is Mover P1) (moveAgain) (remove (last To)))
8.12.5  or

Moves one of the moves in the list.

**Format**

For making a move between two sets of moves.

\[
\text{(or } \text{<moves> <moves> [<then>]} \text{)}
\]

where:

- \text{<moves>: The first move.}
- \text{<moves>: The second move.}
- \text{[<then>]: The moves applied after that move is applied.}

For making a move between many sets of moves.

\[
\text{(or } \{ \text{<moves>} \} [\text{<then>}] \text{)}
\]

where:

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

**Examples**

\[
\text{(or (set Score P1 100) (set Score P2 100))}
\]

\[
\text{(or \{ (set Score P1 100) (set Score P2 100) (set Score P3 100) \})}
\]
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.
9.1 ToBool

9.1.1 toBool

Converts a IntFunction or a FloatFunction to a boolean (false if 0, else true).

Format

\[(\text{toBool} \ (<\text{int} > \ | \ <\text{floatFunction}>))\]

where:
- \(<\text{int}>\): The int function.
- \(<\text{floatFunction}>\): The float function.

Examples

\[(\text{toBool} (\text{count Moves}))\]
\[(\text{toBool} (\text{sqrt 2}))\]
9.2 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.

9.2.1 all

Returns whether all aspects of the specified query are true.

Format

For checking a condition in each value of a integer array.
(all Values <intArrayFunction> if:<boolean>)

where:
- • <intArrayFunction>: The array to check.
- • if:<boolean>: The condition to check.

For checking a condition in each site of a region.
(all <allSitesType> <region> if:<boolean>)

where:
- • <allSitesType>: The query type to perform.
- • <region>: The region to check.
- • if:<boolean>: The condition to check.

For a test with no parameter.
(all <allSimpleType>)

where:
- • <allSimpleType>: The query type to perform.

Examples
(all Values (values Remembered) if:(> 2 (value)))
(all Sites (sites Occupied by:Mover) if:(= 2 (size Stack at:(site))))
(all DiceUsed)
(all Passed)
(all DiceEqual)

9.2.2 allSimpleType
Defines the query types that can be used for an \texttt{(all \ldots)} test with no parameter.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DiceUsed</td>
<td>Returns whether all the dice have been used in the current turn.</td>
</tr>
<tr>
<td>DiceEqual</td>
<td>Returns whether all the dice are equal when they are rolled.</td>
</tr>
<tr>
<td>Passed</td>
<td>Returns whether all players have passed in succession.</td>
</tr>
</tbody>
</table>

9.2.3 allSitesType
Defines the query types that can be used for an \texttt{(all \ldots)} test related to sites.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sites</td>
<td>Returns whether all the sites satisfy a condition.</td>
</tr>
<tr>
<td>Different</td>
<td>Returns whether all the sites are different.</td>
</tr>
</tbody>
</table>
9.3 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.

9.3.1 can

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

Format

(\text{can Move} \ <\text{moves}>)

where:

- \text{<moves>}: List of moves.

Example

(\text{can Move} \ (\text{forEach Piece}))
9.4 DeductionPuzzle

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

9.4.1 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.
9.5 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.

9.5.1 all

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

Format

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

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

Examples

\[(\text{all Different})\]
\[(\text{all Different except:0})\]
9.6 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.

9.6.1 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 [Default SiteType of the board].
  - [<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)

9.6.2 **isPuzzleRegionResultType**

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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>To check if the count of a region is equal to the result.</td>
</tr>
<tr>
<td>Sum</td>
<td>To check if the sum of a region is equal to the result.</td>
</tr>
</tbody>
</table>
9.7 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.

9.7.1 is

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

Format

For checking hidden information at a location for a specific player.

(is Hidden [<hiddenData>] [<siteType>] at:<int> [level:<int>] (to:<player> | to:<roleType>))

where:
- [hiddenData]: The type of hidden data [Invisible].
- [siteType]: The graph element type [default of the board].
- at:<int>: The site to set the hidden information.
- [level:<int>]: The level to set the hidden information [0].
- to:<player>: The player with these hidden information.
- to:<roleType>: The roleType with these hidden information.

For detecting a specific pattern from a site.

(is Repeat [<repetitionType>])

where:
- [repetitionType]: The type of repetition [Positional].

For detecting a specific pattern from a site.

(is Pattern {<stepType>} [<siteType>] [from:<int>] ([what:<int>] | [whats:{<int>}] )

where:
- {stepType}: The walk describing the pattern.
- [siteType]: The type of the site from to detect the pattern.
- [from:<int>]: The site from to detect the pattern [(last To)].
• [what:<int>]: The piece to check in the pattern [piece in from].
• [whats:{<int>}]: The sequence of pieces to check in the pattern [piece in from].

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 recognise the k (in k-regular graph) is odd or not.
• [even:<boolean>]: Flag to recognise 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 parameter.
(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 [default SiteType of the board].
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 [default SiteType of the board].
- [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 [default SiteType of the board].
- <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"].
• \{\texttt{int}\}: The configuration defined by the indices of each piece.
• \[\texttt{int}\]: The specific site of the configuration.
• \[\{\texttt{int}\}\]: The specific sites of the configuration.

For test relative to a connection.
(is \texttt{<isConnectType>} \[\texttt{[<int>] [<siteType>] [at:<int>] [direction]}\] 
\(\{(\texttt{<region>}) | \texttt{<roleType>} | \texttt{<regionTypeStatic>}\}\))

where:
• \texttt{<isConnectType>}: The type of query to perform.
• \texttt{[<int>]}: The minimum number of regions to connect [All of them].
• \texttt{[<siteType>]}: The graph element type [default SiteType of the board].
• \texttt{[at:<int>]}: The specific starting position need to connect.
• \texttt{[direction]}: The directions of the connected pieces used to connect the region [Adjacent].
• \texttt{[region]}: The disjointed regions set, which need to use for connection.
• \texttt{<roleType>}: The role of the player.
• \texttt{<regionTypeStatic>}: Type of the regions to connect.

For test relative to a line.
(is Line \[\texttt{[<siteType>]} \texttt{<int>} [absoluteDirection] [through:<int>] 
[throughAny:<region>] ([roleType] ([what:<int>] | whats:{\texttt{<int>}}) [exact:<boolean>] [contiguous:<boolean>] 
[if:<boolean>] [byLevel:<boolean>]\))

where:
• \texttt{<siteType>}: The graph element type [default SiteType of the board].
• \texttt{<int>}: Minimum length of lines.
• \texttt{[absoluteDirection]}: Direction category to which potential lines must belong [Adjacent].
• \texttt{[through:<int>]}: Location through which the line must pass.
• \texttt{[throughAny:<region>]}: The line must pass through at least one of these sites.
• \texttt{[roleType]}: The owner of the pieces making a line.
• \texttt{[what:<int>]}: The index of the component composing the line.
• \texttt{[whats:{<int>}]}: The indices of the components composing the line.
• \texttt{[exact:<boolean>]}: If true, then lines cannot exceed minimum length [False].
• [contiguous:<boolean>]: If true, the line has to be contiguous [True].
• [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>}])
  [<direction>] [<int>] ([<int>] | [<region>]) [path:<boolean>])

where:
• [<siteType>]: The graph element type [default SiteType of the board].
• [surround:<roleType>]: Used to define the inside condition of the loop.
• [{<roleType>}]: The list of items inside the loop.
• [<direction>]: The direction of the connection [Adjacent].
• [<int>] : The owner of the looping pieces [Mover].
• [<int>]: The starting point of the loop [(last To)].
• [<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>] [from:<int>] (<player> | <roleType>)
  length:<rangeFunction> [closed:<boolean>])

where:
• <siteType>: The graph element type [default SiteType of the board].
• [from:<int>]: The site to look the path [(last To)].
• <player>: The owner of the pieces on the path.
• <roleType>: The role of the player owning the pieces on the path.
• length:<rangeFunction>: The range size of the path.
• [closed:<boolean>]: Is used to detect closed components [False].

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 [default SiteType of the board].
- `<int>`: The index of the site.

For testing if a site is in a region or if an integer is in an array of integers.

\[
\text{is In ([<int>] | [{<int>}]} (<region> (<intArrayFunction>))
\]

where:

- `[<int>]`: The site [(to)].
- `[{<int}>]`: The sites.
- `<region>`: The region.
- `<intArrayFunction>`: The array of integers.

Examples
(is Hidden at:(to) to:Mover)
(is Repeat Positional)
(is Pattern { F R F R F })
(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 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 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:(exact 4))
(is Empty (to))
9.7.2 isComponentType
Defines the types of Is test according to a component and a site/region.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threatened</td>
<td>To check if a location is under threat.</td>
</tr>
<tr>
<td>Within</td>
<td>To check if a specific piece is on the designed region.</td>
</tr>
</tbody>
</table>

9.7.3 isConnectType
Defines the types of Is for a connected or blocked test.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connected</td>
<td>To check if regions are connected by pieces owned by a player.</td>
</tr>
<tr>
<td>Blocked</td>
<td>To check if a player can not connect regions with his pieces.</td>
</tr>
</tbody>
</table>

9.7.4 isGraphType
Defines the types of Is test according to a graph element.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LastFrom</td>
<td>Check the graph element type of the “from” location of the last move.</td>
</tr>
<tr>
<td>LastTo</td>
<td>Check the graph element type of the “to” location of the last move.</td>
</tr>
</tbody>
</table>

9.7.5 isIntegerType
Defines the types of Is test according to an integer.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Odd</td>
<td>To check if a value is odd.</td>
</tr>
<tr>
<td>Even</td>
<td>To check if a value is even.</td>
</tr>
<tr>
<td>Visited</td>
<td>To check if a site was already visited by a piece in the same turn.</td>
</tr>
<tr>
<td>SidesMatch</td>
<td>To detect whether the terminus of a tile matches with its neighbors.</td>
</tr>
<tr>
<td>PipsMatch</td>
<td>To detect whether the pips of a domino match its neighbours.</td>
</tr>
<tr>
<td>Flat</td>
<td>To Ensures that in a 3D board, all the pieces in the bottom layer must be placed so that they do not fall.</td>
</tr>
<tr>
<td>AnyDie</td>
<td>To check if any current die is equal to a specific value.</td>
</tr>
</tbody>
</table>

9.7.6 isPlayerType
Defines the types of Is test for a player.
### 9.7.7 isSimpleType

Defines the types of Is test for a player with no parameter.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle</td>
<td>To check if the game is repeating the same set of states three times with</td>
</tr>
<tr>
<td></td>
<td>exactly the same moves during these states.</td>
</tr>
<tr>
<td>Pending</td>
<td>To check if the state is in pending.</td>
</tr>
<tr>
<td>Full</td>
<td>To check if the board is full.</td>
</tr>
</tbody>
</table>

### 9.7.8 isSiteType

Defines the types of Is test for a site.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>To check if a site is empty.</td>
</tr>
<tr>
<td>Occupied</td>
<td>To check if a site is occupied.</td>
</tr>
</tbody>
</table>

### 9.7.9 isStringType

Defines the types of Is test according to a String parameter.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed</td>
<td>To check if a specific proposition was made.</td>
</tr>
<tr>
<td>Decided</td>
<td>To check if a specific proposition was decided.</td>
</tr>
</tbody>
</table>

### 9.7.10 isTreeType

Defines the types of Is test for a regular graph.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>To check if the induced graph (by adding or deleting edges) is a tree or</td>
</tr>
<tr>
<td></td>
<td>not.</td>
</tr>
<tr>
<td>SpanningTree</td>
<td>To check if the induced graph (by adding or deleting edges) is a spanning</td>
</tr>
<tr>
<td></td>
<td>tree or not.</td>
</tr>
<tr>
<td>Boolean Functions - Is</td>
<td>199</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----</td>
</tr>
<tr>
<td>CaterpillarTree</td>
<td>To check if the induced graph (by adding or deleting edges) is the largest caterpillar Tree or not.</td>
</tr>
<tr>
<td>TreeCentre</td>
<td>To check whether the last vertex is the centre of the tree (or sub tree).</td>
</tr>
</tbody>
</table>
9.8 Math

Math queries return a boolean result based on given inputs.

9.8.1 and

Returns whether all specified conditions are true.

Format

For an and between two booleans.

(\texttt{and } \texttt{<boolean> } \texttt{<boolean>})

where:

- \texttt{<boolean>}: First condition.
- \texttt{<boolean>}: Second condition.

For an and between many booleans.

(\texttt{and } \{\texttt{<boolean>}\})

where:

- \{\texttt{<boolean>}\}: The list of conditions to check.

Examples

(\texttt{and } (= \texttt{(who at:(last To))} \texttt{(mover)}) \texttt{(!=} \texttt{(who at:(last From))} \texttt{(mover)})

(\texttt{and }
  \{
    (= \texttt{(who at:(last To))} \texttt{(mover)})
    \texttt{(!=} \texttt{(who at:(last From))} \texttt{(mover)})
    \texttt{(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.
9.8.2  = (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))

9.8.3  >= (ge)
Tests if valueA >= valueB.

Format

(>= <int> <int>)
where:
• <int>: The left value.
• <int>: The right value.
Example

```
(> (mover) (next))
```

---

### 9.8.4 \( > \) (gt)

Tests if valueA > valueB.

**Format**

```
(> <int> <int>)
```

where:
- `<int>`: The left value.
- `<int>`: The right value.

**Example**

```
(> (mover) (next))
```

---

### 9.8.5 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 boolean returned if the condition is true.
- ` [<boolean>]`: The boolean returned if the condition is false.

**Example**

```
(if (is Mover (next)) (is Pending))
```
9.8.6  \(<=\) (le)
Tests if \(valueA \leq valueB\).

Format

\[(<= <int> <int>)\]

where:
- \(<int>\): The left value.
- \(<int>\): The right value.

Example

\[(<= (mover) (next))\]

9.8.7  \(<\) (lt)
Tests if \(valueA < valueB\).

Format

\[(< <int> <int>)\]

where:
- \(<int>\): The left value.
- \(<int>\): The right value.

Example

\[(< (mover) (next))\]

9.8.8  not
Tests the not condition.
### Boolean Functions - Math Ludii

**Format**

```
(not <boolean>)
```

where:
- **<boolean>**: The condition.

**Example**

```
(not (is In (last To) (sites Mover)))
```

---

**9.8.9  != (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))
```
9.8.10  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)
  }
)

9.8.11  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)))
```
9.9 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.

9.9.1 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)
9.10  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.

9.10.1  was

Returns whether a specified event has occurred in the game.

**Format**

(was Pass)

**Example**

(was Pass)
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 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.
10.1 ToInt

10.1.1 toInt

Converts a BooleanFunction or a FloatFunction to an integer.

Format

\[(\text{toInt} \ (<\text{boolean}> \ | \ <\text{floatFunction}>))\]

where:
- \(<\text{boolean}>\): The boolean function.
- \(<\text{floatFunction}>\): The float function.

Examples

\[(\text{toInt} \ (\text{is Full}))\]
\[(\text{toInt} \ (\text{sqrt 2}))\]
10.2 Board

Board functions return an integer value based on the current board state.

10.2.1 ahead

Returns the site in a given direction from a specified site.

Format

```
(ahead [<siteType>] <int> [steps:<int>] [<direction>])
```

where:
- `<siteType>`: The graph element type [default SiteType of the board].
- `<int>`: Source site.
- `[steps:<int>]`: Distance to travel [1].
- `[<direction>]`: The 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.

10.2.2 centrePoint

Returns the index of the central board site.

Format

```
(centrePoint [<siteType>])
```

where:
- `<siteType>`: The graph element type [default SiteType of the board].
Example

```
(centrePoint)
```

---

### 10.2.3 column

Returns the column number in which a given site lies.

**Format**

```
(column [<siteType>] of:<int>)
```

where:
- ` [<siteType>]`: The graph element type [default SiteType of the board].
- ` of:<int>`: The site to check.

**Example**

```
(column of:(to))
```

**Remarks**

Returns OFF (-1) if the site does not belong to any column.

---

### 10.2.4 coord

Returns the site index of a given board coordinate.

**Format**

For getting a site according to the coordinate.

```
(coord [<siteType>] <string>)
```

where:
- ` [<siteType>]`: The graph element type [default SiteType of the board].
- ` <string>`: The coordinates of the site.
For getting a site according to the row and column indices.

(coord [<siteType>] row:<int> column:<int>)

where:
- [<siteType>]: The graph element type [default SiteType of the board].
- row:<int>: The row index.
- column:<int>: The column index.

Examples

(coord "A1")
(coord row:1 column:5)

10.2.5 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))

10.2.6 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.

10.2.7  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 P1)
(id "Pawn1")

Remarks
To translate a component, a player or a region to an index.

10.2.8 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.

10.2.9 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. These maps are defined in the equipment.

10.2.10 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.

10.2.11 regionSite

Returns one site of a region.
10.2.12  row

Returns the row of a site.

Format

(row [<siteType>] of:<int>)

where:

- [<siteType>]: The graph element type [default SiteType of the board].
- of:<int>: The site to check.

Example

(row of:(to))
10.3 Board - Where

Where functions return an integer value based on the position of a piece.

10.3.1 where

Returns the site (or level) of a piece if it is on the board/site, else OFF (-1).

**Format**

If a piece is on the board, return its site else Off (-1).

(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 [default SiteType of the board].

If a piece is on the board, return its site else Off (-1).

(where <int> [<siteType>])

where:
- `<int>`: The index of the piece.
- `[<siteType>]`: The graph element type [default SiteType of the board].

If a piece is on a site, return its level else Off (-1).

(where Level <string> (<int> | <roleType>) [state:<int>] [<siteType>] at:<int> [fromTop:<boolean>])

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 [default SiteType of the board].
• at:<int>: The site to check.
• [fromTop:<boolean>]: If true, check the stack from the top [True].

If a piece is on a site, return its level else Off (-1).
(\textbf{where} \textbf{Level} \textbf{<int>} [\textbf{<siteType>}] \textbf{at:<int>} [\textbf{fromTop:<boolean>}])

\begin{itemize}
  \item \textbf{<int>}: The index of the piece.
  \item [\textbf{<siteType>}]: The graph element type [default SiteType of the board].
  \item \textbf{at:<int>}: The site to check.
  \item [\textbf{fromTop:<boolean>}]: If true, check the stack from the top [True].
\end{itemize}

\textbf{Examples}

\begin{itemize}
  \item \textbf{(where} "Pawn" Mover)\textbf{)}
  \item \textbf{(where} (what \textbf{at:}(last To)))\textbf{)}
  \item \textbf{(where Level} (what \textbf{at:}(last To)) \textbf{at:}(last To))\textbf{)}
  \item \textbf{(where Level} "Pawn" Mover \textbf{at:}(last To))\textbf{)}
\end{itemize}

\textbf{Remarks}

The name of the piece can be specific without the number on it because the owner is also specified in the ludeme.
10.4 Card

Card functions return an integer value based on the current state of specified Card components.

10.4.1 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))

10.4.2 cardSiteType

Defines the types of properties which can be returned for the Card super ludeme according an index and optionally a level.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
<td>To return the rank of a card.</td>
</tr>
<tr>
<td>Suit</td>
<td>To return the suit of a card.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>TrumpValue</td>
<td>To return the value of the trump of a card.</td>
</tr>
<tr>
<td>TrumpRank</td>
<td>To return the rank of the trump of a card.</td>
</tr>
</tbody>
</table>
10.5 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.

10.5.1 count

Returns the count of the specified property.

Format

For counting according to no parameters or only a graph element type.

(count Stack [<stackDirection>] [<siteType>] (at:<int> | to:<region>)
   [if:<boolean>] [stop:<boolean>])

where:
- [<stackDirection>]: The direction to count in the stack [FromBottom].
- [<siteType>]: The graph element type [default SiteType of the board].
- at:<int>: The site where is the stack.
- to:<region>: The region where are the stacks.
- [if:<boolean>]: The condition to count in the stack [True].
- [stop:<boolean>]: The condition to stop to count in the stack [False].

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 [default SiteType of the board].

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 [default SiteType of the board].
- [in:<region>]: The region to count.
For counting according to a component.
(count \(<\text{countComponentType}\> [\<\text{siteType}\>] (\[\<\text{roleType}\>] | [of:<int>]) [\<\text{string}\>] [in:<region>])

where:
• \(<\text{countComponentType}\>\): The property to count.
• \(<\text{siteType}\>\): The graph element type [default SiteType of the board].
• \(<\text{roleType}\>\): The role of the player [All].
• [of:<int>]: The index of the player.
• \(<\text{string}\>\): 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 \[<\text{siteType}\>] \[<\text{direction}\>] (\[<\text{roleType}\>] | [of:<int>] | [if:<boolean>]) [min:<int>])

where:
• \[<\text{siteType}\>\]: The graph element type [default SiteType of the board].
• \[<\text{direction}\>\]: The directions of the connection between elements in the group [Adjacent].
• \[<\text{roleType}\>\]: The role of the player [All].
• [of:<int>]: The index of the player.
• [if:<boolean>]: The condition on the pieces to include in the group.
• [min:<int>]: Minimum size of each group [0].

For counting elements in a region of liberties.
(count Liberties \[<\text{siteType}\>] [at:<int>] \[<\text{direction}\>] [if:<boolean>])

where:
• \[<\text{siteType}\>\]: The graph element type [default SiteType of the board].
• [at:<int>]: The site to compute the group [(last To)].
• [<direction>]: The type of directions from the site to compute the group [Adjacent].
• [if:<boolean>]: The condition of the members of the group [ (= (mover) (who at:(to)))].

For counting the number of steps between two sites.
(count Steps [<siteType>] [<relationType>] [<step>] <int> (<int> | <region>))

where:
• [<siteType>]: Graph element type [default SiteType of the board].
• [<relationType>]: The relation type of the steps [Adjacent].
• [<step>]: Define a particular step move to step.
• <int>: The first site.
• <int>: The second site.
• <region>: The second region.

For counting the number of steps between two sites.
(count StepsOnTrack ([<roleType>] | [<player>] | [<string>]) [<int>] [<int>])

where:
• [<roleType>]: The role of the owner of the track [Mover].
• [<player>]: The owner of the track [(mover)].
• [<string>]: The name of the track.
• <int>: The first site.
• <int>: The second site.

Examples
(count
    Stack
    FromTop
    at:(last To)
    if: (= (what at:(to) level:(level)) (id "Disc" P1))
    stop: (= (what at:(to) level:(level)) (id "Disc" P2))
)

(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")))
(count StepsOnTrack (last From) (last To))

### 10.5.2 countComponentType
Defines the types of components that can be counted within a game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pieces</td>
<td>Number of pieces on the board (or in hand), per player or over all players.</td>
</tr>
<tr>
<td>Pips</td>
<td>The number of pips showing on all dice, or dice owned by a specified player.</td>
</tr>
</tbody>
</table>

### 10.5.3 countSimpleType
Defines the types of properties that can be counted without a parameter (apart from the graph element type, where relevant).

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>Number of rows on the board.</td>
</tr>
<tr>
<td>Columns</td>
<td>Number of columns on the board.</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Turns</td>
<td>Number of turns played so far in this trial.</td>
</tr>
<tr>
<td>Moves</td>
<td>Number of moves made so far in this trial.</td>
</tr>
<tr>
<td>Trials</td>
<td>Number of completed games within a match.</td>
</tr>
<tr>
<td>MovesThisTurn</td>
<td>Number of moves made so far this turn.</td>
</tr>
<tr>
<td>Phases</td>
<td>Number of phase changes during this trial.</td>
</tr>
<tr>
<td>Vertices</td>
<td>Number of adjacent (connected) elements.</td>
</tr>
<tr>
<td>Edges</td>
<td>Number of edges on the board.</td>
</tr>
<tr>
<td>Cells</td>
<td>Number of cells on the board.</td>
</tr>
<tr>
<td>Players</td>
<td>Number of players.</td>
</tr>
<tr>
<td>Active</td>
<td>Number of active players.</td>
</tr>
</tbody>
</table>

### 10.5.4 countSiteType

Defines the types of sites that can be counted within a game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sites</td>
<td>Number of playable sites within a region or container.</td>
</tr>
<tr>
<td>Adjacent</td>
<td>Number of adjacent (connected) elements.</td>
</tr>
<tr>
<td>Neighbours</td>
<td>Number of neighbours (not necessarily connected).</td>
</tr>
<tr>
<td>Orthogonal</td>
<td>Number of orthogonal elements.</td>
</tr>
<tr>
<td>Diagonal</td>
<td>Number of diagonal elements.</td>
</tr>
<tr>
<td>Off</td>
<td>Number of off-diagonal elements.</td>
</tr>
</tbody>
</table>
10.6 Dice

Dice functions return an integer value based on the current dice roll.

10.6.1 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))
10.7 Iterator

Iterator functions returning an integer value typically used as temporary variables during move planning for chaining nontrivial move sequences or in looping through many values with ludemes such as (forEach ...).

10.7.1 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.

10.7.2 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.

10.7.3 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.

10.7.4 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.

---

**10.7.5  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.

---

**10.7.6  pips**

Returns the number of pips of a die.
10.7.7 player

Returns the “player” value of the context.

**Format**

```
(player)
```

**Example**

```
(player)
```

**Remarks**

This ludeme corresponds to the index of a player. It is used to iterate through the players with a (forEach Player ...) ludeme.

10.7.8 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.

10.7.9  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.

10.7.10  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.
10.8  Last

Last is a ‘super’ ludeme that returns a site related to the last move.

10.8.1  last

Returns a site related to the last move.

**Format**

\[
\text{(last <lastType> [afterConsequence:<boolean>]})
\]

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)

10.8.2  lastType

Defines the types of Last integer ludeme.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>To</td>
<td>To return the “to” site of the last move.</td>
</tr>
<tr>
<td>From</td>
<td>To return the “from” site of the last move.</td>
</tr>
</tbody>
</table>
10.9 Match

Match functions return an integer value based on the state of the current match.

10.9.1 matchScore

Returns the match score of a player.

**Format**

(matchScore <roleType>)

where:

- <roleType>: The roleType of the player.

**Example**

(matchScore P1)
10.10 Math

Math functions return an integer value based on given inputs.

10.10.1 abs

Return the absolute value of a value.

Format

(abs <int>)

where:

- <int>: The value.

Example

(abs (value Piece at:(to)))

10.10.2 + (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.
(+ (list <int> | <intArrayFunction>))

where:

- list <int>: The list of the values.
- <intArrayFunction>: The array of values to sum.
Examples

(+ (value Piece at:(from)) (value Piece at:(to)))
(+
  {
    (value Piece at:(from))
    (value Piece at:(to))
    (value Piece at:(between))
  }
)

10.10.3  / (div)
To divide a value by another.

Format

(/ <int> <int>)
where:
  • <int>: The value to divide.
  • <int>: To divide by b.

Example

(/ (value Piece at:(from)) (value Piece at:(to)))

Remarks
The result will be an integer and round down the result.

10.10.4  if
Returns a value according to a condition.

Format

(if <boolean> <int> <int>)
where:
  • <boolean>: The condition.
• `<int>`: The integer returned if the condition is true.
• `<int>`: The integer 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.

10.10.5 max

Returns the maximum of specified values.

Format

For returning the maximum value between two values.

```
(max <int> <int>)
```

where:

• `<int>`: The first value.
• `<int>`: The second value.

For returning the maximum value between an array of values.

```
(max <intArrayFunction>)
```

where:

• `<intArrayFunction>`: The array of values to maximise.

Examples
(max (mover) (next))

(max
  (results
    from:(last To)
    to:(sites LineOfSight at:(from) All)
    (count Steps All (from) (to))
  )
)

10.10.6  min

Returns the minimum of specified values.

**Format**

For returning the minimum value between two values.
(min <int> <int>)

where:
- <int>: The first value.
- <int>: The second value.

For returning the minimum value between an array of values.
(min <intArrayFunction>)

where:
- <intArrayFunction>: The array of values to minimise.

**Examples**

(min (mover) (next))

(min
  (results
    from:(last To)
    to:(sites LineOfSight at:(from) All)
    (count Steps All (from) (to))
  )
)

(\text{max } (\text{mover} ) (\text{next}))

(\text{max}
  (\text{results}
    \text{from:(last To)}
    \text{to:(sites LineOfSight at:(from) All)}
    (\text{count Steps All (from) (to)})
  )
)
10.10.7  % (mod)
Returns the modulo of a value.

Format

(\% <int> <int>)
where:
  • <int>: The value.
  • <int>: The modulo.

Example

(\% (count Moves) 3)

10.10.8  * (mul)
Returns the product 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>} | <intArrayFunction>)
where:
  • {<int>}: The list of the values.
  • <intArrayFunction>: The array of values to multiply.
Examples

(* (mover) (next))
(* { (mover) (next) (prev) })

10.10.9  \(\wedge\) (pow)
Computes the first parameter to the power of the second parameter.

Format

\((\wedge \ <\text{int}\> \ <\text{int}\>)\)
where:

- \(<\text{int}\>\): The value.
- \(<\text{int}\>\): The power.

Example

\((^ \ (\text{value Piece at:}(\text{last To})) \ 2)\)

10.10.10  \((-\) (sub)
Returns the subtraction \(A\) minus \(B\).

Format

\((- \ [\text{int}] \ <\text{int}\>)\)
where:

- \([\text{int}]\): The first value (to subtract from) \([0]\).
- \(<\text{int}\>\): The second value (to be subtracted from the first value).

Examples

\((- \ 1)\)
\((- \ (\text{value Piece at:}(\text{last To})) \ (\text{value Piece at:}(\text{last From}))\))
# 10.11 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.

## 10.11.1 size

Returns the size of the specified property.

### Format

For the size of a stack.

```
(size Array <intArrayFunction>)
```

where:

- `<intArrayFunction>`: The array.

For the size of a stack.

```
(size Stack [<siteType>] ([in:<region>] | [at:<int>]))
```

where:

- `<siteType>`: The graph element type [default SiteType of the board].
- `[at:<int>]`: The site from which to compute the count [(last To)].

For the size of large pieces currently placed.

```
(size LargePiece [<siteType>] (in:<region> | at:<int>))
```

where:

- `<siteType>`: The graph element type [default site type of the board].
- `[at:<int>]`: The site to look for large piece.

For the size of a group.

```
(size [Group] [siteType] at:<int> [<direction>] [if:<boolean>])
```

where:
• `<siteType>`: The graph element type [default SiteType of the board].
• `at:<int>`: The site to compute the group [(last To)].
• `<direction>`: The type of directions from the site to compute the group [Adjacent].
• `[if:<boolean>]`: The condition of the members of the group [(= (mover) (who at:(to)))].

For the size of a territory.

(size [Territory] [<siteType>] (<roleType> | <player>)
[<absoluteDirection>])

where:
• `<siteType>`: The graph element type [default SiteType of the board].
• `<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 Array (values Remembered))
(size Stack at:(last To))
(size LargePiece at:(last To))
(size Group at:(last To) Orthogonal)
(size Territory P1)
10.12 Stacking

Stacking functions return an integer value based on the state of a specified stack of components.

10.12.1 topLevel

Returns the top level of a stack.

Format

\[(\text{topLevel} \ [\text{<siteType>}] \ at:\text{<int>})\]

where:
- \[\text{<siteType>}\]: The graph element type [default SiteType of the board].
- \[\text{at:<int>}\]: The site of the stack.

Example

\[(\text{topLevel} \ at:\text{(last To)})\]

Remarks

If the game is not a stacking game, then level 0 is returned.
10.13 State

State functions return an integer value based on the current game state.

10.13.1 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)
```

10.13.2 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).

10.13.3 mover

Returns the index of the current player.
10.13.4 next

Returns the index of the next player.

**Format**

(mover)

**Example**

(mover)

**Remarks**

This ludeme is used to apply some specific condition or rule to the next player.

10.13.5 pot

Returns the pot of the game.

**Format**

(pot)
Example

(prev)

10.13.6 prev

Returns the index of the previous player.

Format

(prev [<prevType>])

where:

- [<prevType>]: The type of the previous state [Mover].

Example

(prev)

Remarks

To apply some specific conditions/rules to the previous player.

10.13.7 rotation

Returns the rotation value of a specified site.

Format

(rotation [<siteType>] at:<int> [level:<int>])

where:

- [<siteType>]: The graph element type [default SiteType of the board].
- at:<int>: The location to check.
- [level:<int>]: The level to check [0].
10.13.8  **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)
```

10.13.9  **state**

Returns the local state value of a specified site.

**Format**

```
(state [<siteType>] at:<int> [level:<int>])
```

where:
- `<siteType>`: The graph element type [default SiteType of the board].
- `at:<int>`: The location to check.
- `[level:<int>]`: The level to check [0].

**Example**

```
(state at:(last To))
```
Remarks
This ludeme is used for games with local state values associated with pieces.

10.13.10  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.

10.13.11  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 [default SiteType of the board].
- at:<int>: The location to check.
- [level:<int>]: The level to check [0].

Example
(what at:(last To))
10.13.12 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 [default SiteType of the board].
- `at:<int>`: The location to check.
- `[level:<int>]`: The level to check.

**Example**

```
(who at:(last To))
```
10.14 Tile

Tile functions return an integer value based on the current state of specified Tile components.

10.14.1 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.
10.15 TrackSite

TrackSite is a ‘super’ ludeme that returns a site on a track.

10.15.1 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))
10.16 Value

Value is a ‘super’ ludeme that returns the value of a specified property within the game, such as a player or a component.

10.16.1 value

Returns the value of the specified property.

Format

For returning a random value within a range.
(value Random <rangeFunction>)

where:
- <rangeFunction>: The range.

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 [<siteType>] at:<int> [level:<int>])

where:
- [<siteType>]: The graph element type [default SiteType of the board].
- at:<int>: The location to check.
- [level:<int>]: The level to check.
For returning the value iterated in (forEach Value ...).

(value)

Examples

(value Random (range 3 5))
(value Pending)
(value Player (who at:(to)))
(value Piece at:(to))
(value)
11

Integer Array Functions

Integer array functions are ludemes that return an array of integer values, typically for processing by other ludemes.
11.1 Array

Int array function to convert a region function to an int array.

11.1.1 array

Converts a Region Function to an Int Array.

**Format**

```
(array <region>)
```

where:

- `<region>`: The region function to be converted into an int array.

**Example**

```
(array (sites Board))
```
11.2 Iteraror

Int array iterators.

11.2.1 team

Returns the team iterator.

Format

(team)

Example

(team)
11.3 Math

Math int array functions return an integer of integer values based on given inputs.

11.3.1 difference

Returns the difference between two arrays of integers, i.e. the elements in A that are not in B.

Format

(difference <intArrayFunction> (<intArrayFunction> | <int>))

where:
- <intArrayFunction>: The original array.
- <intArrayFunction>: The array to remove from the original array.
- <int>: The integer to remove from the original array.

Example

(difference (values Remembered) 2)

11.3.2 intersection

Returns the intersection of many regions.

Format

For the intersection of two arrays.
(intersection <intArrayFunction> <intArrayFunction>)

where:
- <intArrayFunction>: The first array.
- <intArrayFunction>: The second array.

For the intersection of many arrays.
(intersection {<intArrayFunction>})

where:
Examples

(intersection (values Remembered "Earth") (values Remembered "Sea"))

(intersection

  (values Remembered "Earth")
  (values Remembered "Sea")
  (values Remembered "Sky")

)

11.3.3 results

Returns an array of all the results of the function for each site ‘from’ to each site ‘to’.

Format

(results (from:<int> | from:<region>) (to:<int> (to:<region>) <int>)

where:

- from:<int>: The ‘from’ site.
- from:<region>: The ‘from’ region.
- to:<int>: The ‘to’ site.
- to:<region>: The ‘to’ region.
- <int>: The function to compute for each site ‘from’ and each site ‘to’.

Example

(results

  from:(last To)
  to:(sites LineOfSight at:(from) All)
  (count Steps All (from) (to))

)

11.3.4 union

Merges many integer arrays into one.
Format

For the union of two arrays.
(union <intArrayFunction> <intArrayFunction>)

where:
- <intArrayFunction>: The first array.
- <intArrayFunction>: The second array.

For the union of many arrays.
(union {<intArrayFunction>})

where:
- {<intArrayFunction>}: The different arrays.

Examples

(union (values Remembered "Forest") (values Remembered "Sea"))

(union
  { 
    (values Remembered "Forest")
    (values Remembered "Sea")
    (values Remembered "Sky")
  } )
11.4 Players

Int array functions returning player indices.

11.4.1 players

Returns an array of players indices.

**Format**

For returning the indices of the players in a team.

```
(players <playersTeamType> [if:<boolean>])
```

where:
- `<playersTeamType>`: The player type to return.
- `[if:<boolean>]`: The condition to keep the players [True].

For returning the indices of players related to others.

```
(players <playersManyType> [of:<int>] [if:<boolean>])
```

where:
- `<playersManyType>`: The player type to return.
- `[of:<int>]`: The index of the related player.
- `[if:<boolean>]`: The condition to keep the players.

**Examples**

```
(players Team1)
(players Team1)
```

11.4.2 playersManyType

Defines the types of set of players which can be iterated.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All players.</td>
</tr>
<tr>
<td>NonMover</td>
<td>Players who are not moving.</td>
</tr>
<tr>
<td>Enemy</td>
<td>Enemy players.</td>
</tr>
<tr>
<td>Friend</td>
<td>Friend players (Mover + Allies).</td>
</tr>
</tbody>
</table>
Ally

### 11.4.3 playersTeamType

Defines the types of team which can be iterated.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Team1</td>
<td>Team 1.</td>
</tr>
<tr>
<td>Team2</td>
<td>Team 2.</td>
</tr>
<tr>
<td>Team3</td>
<td>Team 3.</td>
</tr>
<tr>
<td>Team4</td>
<td>Team 4.</td>
</tr>
<tr>
<td>Team5</td>
<td>Team 5.</td>
</tr>
<tr>
<td>Team6</td>
<td>Team 6.</td>
</tr>
<tr>
<td>Team7</td>
<td>Team 7.</td>
</tr>
<tr>
<td>Team8</td>
<td>Team 8.</td>
</tr>
<tr>
<td>Team9</td>
<td>Team 9.</td>
</tr>
<tr>
<td>Team10</td>
<td>Team 10.</td>
</tr>
<tr>
<td>Team11</td>
<td>Team 11.</td>
</tr>
<tr>
<td>Team12</td>
<td>Team 12.</td>
</tr>
<tr>
<td>Team13</td>
<td>Team 13.</td>
</tr>
<tr>
<td>Team14</td>
<td>Team 14.</td>
</tr>
<tr>
<td>Team15</td>
<td>Team 15.</td>
</tr>
<tr>
<td>Team16</td>
<td>Team 16.</td>
</tr>
</tbody>
</table>
11.5 Sizes

Int array functions return sizes of regions.

11.5.1 sizes

Returns an array of sizes of many regions.

**Format**

\[
\text{(sizes Group } [<\text{siteType}>] \ [direction>] \ ([<\text{roleType}>] | [of:<int>] | [if:<boolean>]) \ [min:<int>])
\]

where:
- \([<\text{siteType}>]\): The graph element type [default SiteType of the board].
- \([direction>]\): The directions of the connection between elements in the group [Adjacent].
- \([<\text{roleType}>]\): The role of the player [All].
- \([of:<int>]\): The index of the player.
- \([if:<boolean>]\): The condition on the pieces to include in the group.
- \([min:<int>]\): Minimum size of each group [0].

**Example**

\[
\text{(sizes Group Orthogonal P1)}
\]
11.6 **State**

State int array functions return integer values based on the current game state.

### 11.6.1 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)
```
11.7 Values

State int array functions return integer values stored in the state after remembering them.

11.7.1 values

Returns an array of values.

Format

(values Remembered [<string>])

where:
- [<string>]: The name of the remembering values.

Example

(values Remembered)
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.
12.1 Foreach

To iterate sites or players to compute sites.

12.1.1 forEach

Returns a region filtering with a condition or build according to different player indices.

**Format**

For iterating through teams.

```
(forEach Team <region>)
```

where:
- `<region>`: The region.

For filtering a region according to a condition.

```
(forEach <region> if:<boolean>)
```

where:
- `<region>`: The original region.
- `if:<boolean>`: The condition to satisfy.

For iterating on players.

```
(forEach <intArrayFunction> <region>)
```

where:
- `<intArrayFunction>`: The list of players.
- `<region>`: The region.

**Examples**

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

```
(forEach (sites Occupied by:P1) if:=(what at:(site)) (id "Pawn1"))
```

```
(forEach (players Ally of:(next)) (sites Occupied by:Player))
```
12.2 Last

Last region functions returning region of sites based on the previous state.

12.2.1 last

Returns sites related to the last move.

Format

(last Between)

Example

(last Between)
12.3 Math

Math region functions return a combined region of sites based on provided input regions.

12.3.1 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))
```

12.3.2 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 [default SiteType of the board].
Example

(expand (sites Bottom) steps:2)

12.3.3 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))

12.3.4 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:
  • \{<\text{region}>\}: The different regions.

Examples

\[(\text{intersection} \text{ (sites Mover) (sites Occupied by:Mover)})\]
\[(\text{intersection} \{ \text{ (sites Mover) (sites Occupied by:Mover) (sites Occupied by:Next) } \})\]

\subsection{12.3.5 \underline{union}}

Merges many regions into one.

Format

For the union of two regions.
\[(\text{union} <\text{region}> <\text{region}>)\]
where:
  • \text{ <region>}: The first region.
  • <region>: The second region.

For the union of many regions.
\[(\text{union} \{<\text{region}>\})\]
where:
  • \{<\text{region}>\}: The different regions.

Examples

\[(\text{union} \text{ (sites P1) (sites P2)})\]
\[(\text{union} \{ \text{ (sites P1) (sites P2) (sites P3) } \})\]
12.4 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.

12.4.1 lineOfSightType

Specifies the expected types of line of sight tests.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>Empty sites in line of sight along each direction.</td>
</tr>
<tr>
<td>Farthest</td>
<td>Farthest empty site in line of sight along each direction.</td>
</tr>
<tr>
<td>Piece</td>
<td>First piece (of any type) in line of sight along each direction.</td>
</tr>
</tbody>
</table>

12.4.2 sites

Returns the specified set of sites.

Format

For getting the sites in a loop or making the loop.

(sites Loop [inside:<boolean>] [<siteType>] ([surround:<roleType>] | [{<roleType>}] [direction] [int] ([int] | [region]))

where:
- [inside:<boolean>]: True to return the sites inside the loop [False].
- [<siteType>]: The graph element type [default SiteType of the board].
- [surround:<roleType>]: Used to define the inside condition of the loop.
- [{<roleType}>]: The list of items inside the loop.
- [direction]: The direction of the connection [Adjacent].
- [int]: The owner of the looping pieces [Mover].
- [int]: The starting point of the loop [(last To)].
- [region]: The region to start to detect the loop.

For getting the sites in a pattern.

(sites Pattern {<stepType>} [<siteType>] [from:<int>] ([what:<int>] | [whats:{<int>}])))

where:
Region Functions - Sites Ludii

- \{<\text{stepType}>\}: The walk describing the pattern.
- \[<\text{siteType}>\]: The type of the site from to detect the pattern.
- \[\text{from:<int>}\]: The site from to detect the pattern ([last To]).
- \[\text{what:<int>}\]: The piece to check in the pattern [piece in from].
- \[\text{whats:{<int>}\}]: The sequence of pieces to check in the pattern [piece in from].

For getting the sites with specific hidden information for a player.

(sites Hidden \[<\text{hiddenData}>\] \[<\text{siteType}>\] (to:<\text{player}> | to:<\text{roleType}>))

where:
- \[<\text{hiddenData}>\]: The type of hidden data [Invisible].
- \[<\text{siteType}>\]: The graph element type [default of the board].
- to:<\text{player}>: The player with these hidden information.
- to:<\text{roleType}>: The roleType with these hidden information.

For getting the sites iterated in ForEach Moves.

(sites)

For getting the sites (in the same radial) between two others sites.

(sites Between \[<\text{direction}>\] \[<\text{siteType}>\] from:<\text{int>} [fromIncluded:<boolean>] to:<\text{int}> [toIncluded:<boolean>] [cond:<boolean>])

where:
- \[<\text{direction}>\]: The directions of the move [Adjacent].
- \[<\text{siteType}>\]: The type of the graph element [Default SiteType].
- from:<\text{int}>: The 'from' site.
- [fromIncluded:<boolean>]: True if the 'from' site is included in the result [False].
- to:<\text{int}>: The 'to' site.
- [toIncluded:<boolean>]: True if the 'to' site is included in the result [False].
- [cond:<boolean>]: The condition to include the site in between [True].
For getting the sites occupied by a large piece from the root of the large piece.
(sites LargePiece [siteType] at:int)
where:
  • [siteType]: The type of the graph element [Default SiteType].
  • at:int: The site to look (the root).

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 | from:region) [direction] [if:boolean])
where:
  • [siteType]: The type of the graph elements of the group.
  • at:int: The specific starting position of the group.
  • from:region: The specific starting positions of the groups.
  • [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.

\[(\text{sites } \langle \text{sitesEdgeType} \rangle)\]

where:

- \(\langle \text{sitesEdgeType} \rangle\): Type of sites to return.

For getting sites without any parameter or only the graph element type.

\[(\text{sites } \langle \text{sitesSimpleType} \rangle [\langle \text{siteType} \rangle])\]

where:

- \(\langle \text{sitesSimpleType} \rangle\): Type of sites to return.
- \([\langle \text{siteType} \rangle]\): The graph element type [default SiteType of the board].

For getting sites according to their coordinates.

\[(\text{sites } [\langle \text{siteType} \rangle \{\langle \text{string} \rangle\})\]

where:

- \([\langle \text{siteType} \rangle]\): The graph element type [default SiteType of the board].
- \(\{\langle \text{string} \rangle\}\): 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.

\[(\text{sites } \langle \text{sitesMoveType} \rangle \langle \text{moves} \rangle)\]

where:

- \(\langle \text{sitesMoveType} \rangle\): Type of sites to return.
- \(\langle \text{moves} \rangle\): The moves for which to collect the positions.

For creating a region from a list of site indices or an IntArrayFunction.

\[(\text{sites } (\{\langle \text{int} \rangle\} | \langle \text{IntArrayFunction} \rangle))\]

where:

- \(\{\langle \text{int} \rangle\}\): The list of the sites.
- \(\langle \text{IntArrayFunction} \rangle\): The IntArrayFunction.
For getting sites of a walk.

\[
\text{(sites } [\text{siteType}] [\text{int}] \{\{\text{stepType}\}\} \text{ [rotations: boolean]})
\]

where:
- \[\text{siteType}\]: The graph element type [default SiteType of the board].
- \[\text{int}\]: The location from which to compute the walk [from].
- \[\{\text{stepType}\}\]: The different turtle steps defining a graphic turtle walk.
- \[\text{rotations: boolean}\]: True if the move includes all the rotations of the walk [True].

For getting sites belonging to a part of the board.

\[
\text{(sites } <\text{sitesIndexType}> [\text{siteType}] [\text{int}])
\]

where:
- \[<\text{sitesIndexType}>\]: Type of sites to return.
- \[<\text{siteType}>\]: The graph element type [default SiteType of the board].
- \[<\text{int}cea\]: 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.

\[
\text{(sites Side } <\text{siteType}> ([<\text{player}>] | <\text{roleType}>) | <\text{compassDirection}>))
\]

where:
- \[<\text{siteType}>\]: The graph element type [default SiteType of the board].
- \[<\text{player}>\]: Index of the player or the component.
- \[<\text{roleType}>\]: The Role type corresponding to the index.
- \[<\text{compassDirection}>\]: Direction of the side to return.

For getting sites at a specific distance of another.

\[
\text{(sites Distance } <\text{siteType}> [<\text{relationType}>] [<\text{step}>] \text{ from: int } \text{ rangeFunction})
\]

where:
• [<siteType>]: The graph element type [default site type of the board].
• [<relationType>]: The relation type of the steps [Adjacent].
• [<step>]: Define a particular step move to step.
• from:<int>: Index of the site.
• <rangeFunction>: 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 track.
(sites Track ([<player>] | [<roleType>]) [<string>] [from:<int>] [to:<int>])

where:
• [<player>]: Index of the player owned the track.
• [<roleType>]: The Role type corresponding to the index.
• [<string>]: The name of the track.
• [from:<int>]: Only the sites in the track from that site (included).
• [to:<int>]: Only the sites in the track until to reach that site (included).

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 [default SiteType].
• [<player>]: Index of the player or the component.
• \[<\text{roleType}>\]: The Role type corresponding to the index.
• \[<\text{nonDecision}>\]: Rules used to generate moves for finding winning sites.
• \[<\text{string}>\]: The name of the board or region to return.

For getting sites relative to a player.
(sites Start \(<\text{piece}>\))

where:
• \(<\text{piece}>\): Index of the player or the component.

For getting sites occupied by player(s).
(sites Occupied (by:<\text{player}> | by:<\text{roleType}>) ([container:<int>]
  ([container:<string>]) ([component:<int>]) ([component:<string>] | [
  components:{<string>}] [top:<boolean>] [on:<\text{siteType}>]))

where:
• by:<\text{player}>: The index of the owner.
• by:<\text{roleType}>: The roleType of the owner.
• [container:<int>]: The index of the container.
• [container:<string>]: The name of the container.
• [component:<int>]: The index 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].
• [on:<\text{siteType}>]: The type of the graph element.

For getting sites incident to another.
(sites Incident \(<\text{siteType}>\) of:<\text{siteType}> at:<int> ([owner:<\text{player}>] | [roleType]))

where:
• \(<\text{siteType}>\): The graph type of the result.
• of:<\text{siteType}>: The graph type of the index.
• at:<int>: Index of the element to check.
• [owner:<\text{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 [default SiteType of the board].
  • `<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>] [stopIncluded:<boolean>]
  [distance:<int>] `<siteType`)]

where:
  • from:<int>`: The origin 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].
  • [stopIncluded:<boolean>]: True if the site stopping the radial in each direction is included in the result [False].
  • [distance:<int>]: The distance around which to check [Infinite].
  • `<siteType>`: The graph element type [default SiteType of the board].
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 [default SiteType of the board].
- [at:<int>]: The location [(last To)].
- [<direction>]: The directions of the move [Adjacent].

Examples
(sites Loop)
(sites Pattern { F R F R F })
(sites Hidden to:Mover)
(sites Hidden What to:(player (next)))
(sites Hidden Rotation Vertex to:Next)
(sites)
(sites Between from:(last From) to:(last To))
(sites LargePiece at:(last To))
(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) (exact 5))
(sites P1)
(sites "E5")
(sites Track)
(sites Hand Mover)
(sites Winning Next (add (to (sites Empty))))
(sites Start (piece (what at:(from))))
(sites Occupied by:Mover)
12.4.3 sitesEdgeType
Specifies set of edge sites.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axial</td>
<td>The axials edges sites.</td>
</tr>
<tr>
<td>Horizontal</td>
<td>The horizontal edges sites.</td>
</tr>
<tr>
<td>Vertical</td>
<td>The vertical edges sites.</td>
</tr>
<tr>
<td>Angled</td>
<td>The angled edges sites.</td>
</tr>
<tr>
<td>Slash</td>
<td>The slash edges sites.</td>
</tr>
<tr>
<td>Slosh</td>
<td>The slosh edges sites.</td>
</tr>
</tbody>
</table>

12.4.4 sitesIndexType
Specifies sets of board sites by some indexed property.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row</td>
<td>Sites in a specified row.</td>
</tr>
<tr>
<td>Column</td>
<td>Sites in a specified column.</td>
</tr>
<tr>
<td>Phase</td>
<td>Sites in a specified phase.</td>
</tr>
<tr>
<td>Cell</td>
<td>Vertices that make up a cell.</td>
</tr>
<tr>
<td>Edge</td>
<td>End points of an edge.</td>
</tr>
<tr>
<td>State</td>
<td>Sites with a specified state value.</td>
</tr>
<tr>
<td>Empty</td>
<td>Empty (i.e. unoccupied) sites of a container.</td>
</tr>
<tr>
<td>Layer</td>
<td>Sites in a specified layer.</td>
</tr>
</tbody>
</table>

12.4.5 sitesMoveType
Specifies sets of sites based on the positions of moves.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>From-positions of a collection of moves as a set of sites.</td>
</tr>
<tr>
<td>Between</td>
<td>Between-positions of a collection of moves as a set of sites.</td>
</tr>
<tr>
<td>To</td>
<td>To-positions of a collection of moves as a set of sites.</td>
</tr>
</tbody>
</table>

12.4.6 sitesPlayerType
Specifies sets of sites associated with given players.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
Hand | Sites in a player’s hand.
Winning | Sites that would be winning moves for the current player.

### 12.4.7 sitesSimpleType

Specifies set of sites that do not require any parameters (apart from the graph element type).

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board</td>
<td>All board sites.</td>
</tr>
<tr>
<td>Top</td>
<td>Sites on the top side of the board.</td>
</tr>
<tr>
<td>Bottom</td>
<td>Sites on the bottom side of the board.</td>
</tr>
<tr>
<td>Left</td>
<td>Sites on the left side of the board.</td>
</tr>
<tr>
<td>Right</td>
<td>Sites on the right side of the board.</td>
</tr>
<tr>
<td>Inner</td>
<td>Interior board sites.</td>
</tr>
<tr>
<td>Outer</td>
<td>Outer board sites.</td>
</tr>
<tr>
<td>Perimeter</td>
<td>Perimeter board sites.</td>
</tr>
<tr>
<td>Corners</td>
<td>Corner board sites.</td>
</tr>
<tr>
<td>ConcaveCorners</td>
<td>Concave corner board sites.</td>
</tr>
<tr>
<td>ConvexCorners</td>
<td>Convex corner board sites.</td>
</tr>
<tr>
<td>Major</td>
<td>Major generator board sites.</td>
</tr>
<tr>
<td>Minor</td>
<td>Minor generator board sites.</td>
</tr>
<tr>
<td>Centre</td>
<td>Centre board site(s).</td>
</tr>
<tr>
<td>Hint</td>
<td>Sites that contain a puzzle hint.</td>
</tr>
<tr>
<td>ToClear</td>
<td>Sites to remove at the end of a capture sequence.</td>
</tr>
<tr>
<td>LineOfPlay</td>
<td>Sites in the line of play. Applies to domino game (returns an empty region for other games).</td>
</tr>
<tr>
<td>Pending</td>
<td>Sites with a non-zero “pending” value in the game state.</td>
</tr>
<tr>
<td>Playable</td>
<td>Playable sites of a boardless game. For other games, returns the set of empty sites adjacent to occupied sites.</td>
</tr>
<tr>
<td>LastTo</td>
<td>The set of “to” sites of the last move.</td>
</tr>
<tr>
<td>LastFrom</td>
<td>The set of “from” sites of the last move.</td>
</tr>
</tbody>
</table>
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.
13.1 Difference

The base directions function converts input directions to player-centric equivalents.

13.1.1 difference

Returns the difference of two set of directions.

Format

(difference <direction1> <direction2>)

where:
- <direction1>: The original directions.
- <direction2>: The directions to remove.

Example

(difference Orthogonal N)
13.2 Directions

The base directions function converts input directions to player-centric equivalents.

13.2.1 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].

For returning a direction between two sites of the same type.

(directions <siteType> from:<int> to:<int>)

where:
- <siteType>: The type of the graph element.
- from:<int>: The 'from' site.
- to:<int>: The 'to' site.
For returning a direction between two sites of the same type.

(directions Random <direction> num:<int>)

where:
- <direction>: The direction function.
- num:<int>: The number of directions to return (if possible).

Examples

(directions Orthogonal)
(directions Forwards)
(directions Cell from:(last To) to:(last From))
(directions Random Orthogonal num:2)
13.3 If

The base `directions` function converts input directions to player-centric equivalents.

13.3.1 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)
```
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.
14.1 Range

The base range function defines a range with upper and lower board (inclusive), optionally according to some specified condition.

14.1.1 range

Returns a range of values (inclusive) according to some specified condition.

Format

(range <int> [<int>])

where:
- <int>: Lower extent of range (inclusive).
- [<int>]: Upper extent of range (inclusive) [same as min].

Example

(range (from) (to))
14.2 Math

Math range functions return a range based on specified inputs.

14.2.1 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.

14.2.2 max

Returns a range with a specified maximum (inclusive).

**Format**

```
(max <int>)
```

where:

- `<int>`: Upper extent of range (inclusive).

**Example**

```
(max 4)
```
14.2.3  min

Returns a range with a specified minimum (inclusive).

**Format**

```
(min <int>)
```

where:

- `<int>`: Lower extent of range (inclusive).

**Example**

```
(min 4)
```
Utilities

Utilities ludemes are useful support classes used by various other types of ludemes.

15.1 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.

15.1.1 absoluteDirection

Describes categories of absolute directions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All directions.</td>
</tr>
<tr>
<td>Angled</td>
<td>Angled directions.</td>
</tr>
<tr>
<td>Adjacent</td>
<td>Adjacent directions.</td>
</tr>
<tr>
<td>Axial</td>
<td>Axial directions.</td>
</tr>
<tr>
<td>Orthogonal</td>
<td>Orthogonal directions.</td>
</tr>
<tr>
<td>Diagonal</td>
<td>Diagonal directions.</td>
</tr>
<tr>
<td>OffDiagonal</td>
<td>Off-diagonal directions.</td>
</tr>
<tr>
<td>SameLayer</td>
<td>Directions on the same layer.</td>
</tr>
<tr>
<td>Upward</td>
<td>Upward directions.</td>
</tr>
<tr>
<td>Direction</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Downward</td>
<td>Downward directions.</td>
</tr>
<tr>
<td>Rotational</td>
<td>Rotational directions.</td>
</tr>
<tr>
<td>Base</td>
<td>Base directions.</td>
</tr>
<tr>
<td>Support</td>
<td>Support directions.</td>
</tr>
<tr>
<td>N</td>
<td>North.</td>
</tr>
<tr>
<td>E</td>
<td>East.</td>
</tr>
<tr>
<td>S</td>
<td>South.</td>
</tr>
<tr>
<td>W</td>
<td>West.</td>
</tr>
<tr>
<td>NE</td>
<td>North-East.</td>
</tr>
<tr>
<td>SE</td>
<td>South-East.</td>
</tr>
<tr>
<td>NW</td>
<td>North-West.</td>
</tr>
<tr>
<td>SW</td>
<td>South-West.</td>
</tr>
<tr>
<td>NNW</td>
<td>North-North-West.</td>
</tr>
<tr>
<td>WNW</td>
<td>West-North-West.</td>
</tr>
<tr>
<td>WSW</td>
<td>West-South-West.</td>
</tr>
<tr>
<td>SSW</td>
<td>South-South-West.</td>
</tr>
<tr>
<td>SSE</td>
<td>South-South-East.</td>
</tr>
<tr>
<td>ESE</td>
<td>East-South-East.</td>
</tr>
<tr>
<td>ENE</td>
<td>East-North-East.</td>
</tr>
<tr>
<td>NNE</td>
<td>North-North-East.</td>
</tr>
<tr>
<td>CW</td>
<td>Clockwise directions.</td>
</tr>
<tr>
<td>CCW</td>
<td>Counter-Clockwise directions.</td>
</tr>
<tr>
<td>In</td>
<td>Inwards directions.</td>
</tr>
<tr>
<td>Out</td>
<td>Outwards directions.</td>
</tr>
<tr>
<td>U</td>
<td>Upper direction.</td>
</tr>
<tr>
<td>UN</td>
<td>Upwards-North direction.</td>
</tr>
<tr>
<td>UNE</td>
<td>Upwards-North-East direction.</td>
</tr>
<tr>
<td>UE</td>
<td>Upwards-East direction.</td>
</tr>
<tr>
<td>USE</td>
<td>Upwards-South-East direction.</td>
</tr>
<tr>
<td>US</td>
<td>Upwards-South direction.</td>
</tr>
<tr>
<td>USW</td>
<td>Upwards-South-West direction.</td>
</tr>
<tr>
<td>UW</td>
<td>Upwards-West direction.</td>
</tr>
<tr>
<td>UNW</td>
<td>Upwards-North-West direction.</td>
</tr>
<tr>
<td>D</td>
<td>Down direction.</td>
</tr>
<tr>
<td>DN</td>
<td>Down-North direction.</td>
</tr>
<tr>
<td>DNE</td>
<td>Down-North-East.</td>
</tr>
<tr>
<td>DE</td>
<td>Down-East direction.</td>
</tr>
</tbody>
</table>
DSE | Down-South-East.
---|---
DS | Down-South direction.
DSW | Down-South-West.
DW | Down-West direction.
DNW | Down North West.

### 15.1.2 compassDirection

Compass directions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>North.</td>
</tr>
<tr>
<td>NNE</td>
<td>North-North-East.</td>
</tr>
<tr>
<td>NE</td>
<td>North-East.</td>
</tr>
<tr>
<td>ENE</td>
<td>East-North-East.</td>
</tr>
<tr>
<td>E</td>
<td>East.</td>
</tr>
<tr>
<td>ESE</td>
<td>East-South-East.</td>
</tr>
<tr>
<td>SE</td>
<td>South-East.</td>
</tr>
<tr>
<td>SSE</td>
<td>South-South-East.</td>
</tr>
<tr>
<td>S</td>
<td>South.</td>
</tr>
<tr>
<td>SSW</td>
<td>South-South-West.</td>
</tr>
<tr>
<td>SW</td>
<td>South-West.</td>
</tr>
<tr>
<td>WSW</td>
<td>West-South-West.</td>
</tr>
<tr>
<td>W</td>
<td>West.</td>
</tr>
<tr>
<td>WNW</td>
<td>West-North-West.</td>
</tr>
<tr>
<td>NW</td>
<td>North-West.</td>
</tr>
<tr>
<td>NNW</td>
<td>North-North-West.</td>
</tr>
</tbody>
</table>

### 15.1.3 relativeDirection

Describes categories of relative directions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Forward (only) direction.</td>
</tr>
<tr>
<td>Backward</td>
<td>Backward (only) direction.</td>
</tr>
<tr>
<td>Rightward</td>
<td>Rightward (only) direction.</td>
</tr>
<tr>
<td>Leftward</td>
<td>Leftward (only) direction.</td>
</tr>
<tr>
<td>Forwards</td>
<td>Forwards directions.</td>
</tr>
<tr>
<td>Backwards</td>
<td>Backwards directions.</td>
</tr>
<tr>
<td>Rightwards</td>
<td>Rightwards directions.</td>
</tr>
</tbody>
</table>
### Leftwards
- **FL**: Forward-Left direction.
- **FLL**: Forward-Left-Left direction.
- **FLLL**: Forward-Left-Left-Left direction.

### Backward-Left
- **BL**: Backward-Left direction.
- **BLL**: Backward-Left-Left direction.
- **BLLL**: Backward-Left-Left-Left direction.

### Forward-Right
- **FR**: Forward-Right direction.
- **FRR**: Forward-Right-Right direction.
- **FRRR**: Forward-Right-Right-Right direction.

### Backward-Right
- **BR**: Backward-Right direction.
- **BRR**: Backward-Right-Right direction.
- **BRRR**: Backward-Right-Right-Right direction.

### OppositeDirection
- **SameDirection**: Same direction.
- **OppositeDirection**: Opposite direction.

---

### 15.1.4 stackDirection

Describes the bottom or the top of a stack as origin for functions.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FromBottom</td>
<td>To check the stack from the bottom.</td>
</tr>
<tr>
<td>FromTop</td>
<td>To check the stack from the top.</td>
</tr>
</tbody>
</table>
15.2 End

This section describes support utility ludemes relevant to end rules.

15.2.1 payoff

Defines a payoff to set when using the (payoffs ...) end rule.

Format

```
(payoff <roleType> <floatFunction>)
```

where:

- `<roleType>`: The role of the player.
- `<floatFunction>`: The payoff of the player.

Example

```
(payoff P1 5.5)
```

15.2.2 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)
```
15.3 Equipment

Ludeme utilities are useful support classes that various types of ludeme classes refer to.

15.3.1 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)

15.3.2 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.

\[ (\text{hint int [int]}) \]

where:

- \textit{int}: The location.
- \textit{[int]}: The value of the hint [0].

**Examples**

\[ (\text{hint \{0 1 2 3 4 5\} 1}) \]

\[ (\text{hint 1 1}) \]

**Remarks**

This is used only for deduction puzzles.

### 15.3.3 region

Defines a region of sites within a container.

### 15.3.4 trackStep

Defines a step within a track.

**Format**

\[ (\text{trackStep (int \mid \text{<compassDirection>} \mid \text{<trackStepType>})}) \]

where:

- \textit{int}: Dim function or integer.
- \textit{<compassDirection>}: Compass direction.
- \textit{<trackStepType>}: Track step type: Off/End/Repeat.

**Remarks**

Track steps may be specified by number, dim function, compass direction or constant attribute (End/Off/Repeat).
15.3.5 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))
15.4 Graph

Graph utilities are support classes for describing the graph that defines a game board.

15.4.1 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 } }
)

15.4.2 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.
15.5 Math

Math utilities are support classes for various numerical ludemes.

15.5.1 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 ...).

15.5.2 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:
  • \(<\text{roleType}\rangle\): The key of the pair.
  • \(<\text{int}\rangle\): The corresponding value.

For a pair of two RoleTypes.
(pair \(<\text{roleType}\rangle <\text{roleType}\rangle\)
where:
  • \(<\text{roleType}\rangle\): The key of the pair.
  • \(<\text{roleType}\rangle\): The corresponding value.

For a pair of two strings.
(pair \(<\text{string}\rangle <\text{string}\rangle\)
where:
  • \(<\text{string}\rangle\): The key of the pair.
  • \(<\text{string}\rangle\): The corresponding value.

For a pair of an integer and a string.
(pair \(<\text{int}\rangle <\text{string}\rangle\)
where:
  • \(<\text{int}\rangle\): The key of the pair.
  • \(<\text{string}\rangle\): The corresponding value.

For a pair of a RoleType and a string.
(pair \(<\text{roleType}\rangle <\text{string}\rangle\)
where:
  • \(<\text{roleType}\rangle\): The key of the pair.
  • \(<\text{string}\rangle\): 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.
15.6 Moves

Moves utilities are support classes for defining and generating legal moves.

15.6.1 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))))
```

15.6.2 flips

Sets the flips state of a piece.

**Format**

```
(flips int int)
```

where:
- **int**: The first state of the flip.
- **int**: The second state of the flip.
Example

(flips 1 2)

---

15.6.3 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))

---

15.6.4 piece

Specifies operations based on the “what” data.

Format

(piece (<string> | <int> | {<string> | <int>} | {<string>}) [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> | <int>}: The names of the components.
- {<int>}: The indices of the components.
• \texttt{[state:<int>]}: The local state value to put on the site where the piece is placed.

Example

\begin{verbatim}
(piece (mover))
\end{verbatim}

15.6.5 player

Specifies operations based on the “who” data.

Format

\begin{verbatim}
(player <int>)
\end{verbatim}

where:
\begin{itemize}
  \item \texttt{<int>}: The index of the player \texttt{(mover)}.
\end{itemize}

Examples

\begin{verbatim}
(player (mover))
(player 2)
\end{verbatim}

15.6.6 to

Specifies operations based on the “to” location.

Format

\begin{verbatim}
(to [<siteType>] [[<region>] | [<int>]] [level:<int>] [<rotations>] [if:<boolean>] [apply])
\end{verbatim}

where:
\begin{itemize}
  \item \texttt{[<siteType>]}: The graph element type.
  \item \texttt{[<region>]}: The region of “to” the location.
  \item \texttt{[<int>]}: The “to” location.
  \item \texttt{[level:<int>]}: The level of the “to” location.
  \item \texttt{[<rotations>]}: Rotations of the “to” location.
  \item \texttt{[if:<boolean>]}: The condition on the “to” location.
\end{itemize}
• \(<\text{apply}>\): Effect to apply to the “to” location.

Example

(to (last To) level:(level))
This package defines various types used to specify the behaviour of ludemes. Types are constant values denoted in UpperCamelCase syntax.

16.1 Board

Board types are constant values for specifying various aspects of the board and its constituent graph elements.

16.1.1 basisType

Defines known tiling types for boards.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoBasis</td>
<td>No tiling; custom graph.</td>
</tr>
<tr>
<td>Triangular</td>
<td>Triangular tiling.</td>
</tr>
<tr>
<td>Square</td>
<td>Square tiling.</td>
</tr>
<tr>
<td>Hexagonal</td>
<td>Hexagonal tiling.</td>
</tr>
<tr>
<td>T33336</td>
<td>Semi-regular tiling made up of hexagons surrounded by triangles.</td>
</tr>
<tr>
<td>T33344</td>
<td>Semi-regular tiling made up of alternating rows of squares and triangles.</td>
</tr>
<tr>
<td>T33434</td>
<td>Semi-regular tiling made up of squares and pairs of triangles.</td>
</tr>
<tr>
<td>T3464</td>
<td>Rhombitrihexahedral tiling (e.g. Kensington).</td>
</tr>
<tr>
<td>T3636</td>
<td>Semi-regular tiling 3.6.3.6 made up of hexagons with interstitial triangles.</td>
</tr>
<tr>
<td>T4612</td>
<td>Semi-regular tiling made up of squares, hexagons and dodecagons.</td>
</tr>
<tr>
<td>T488</td>
<td>Semi-regular tiling 4.8.8. made up of octagons with interstitial squares.</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>T31212</td>
<td>Semi-regular tiling made up of triangles and dodecagons.</td>
</tr>
<tr>
<td>T333333,33434</td>
<td>Tiling 3.3.3.3.3.3.3.4.3.4.</td>
</tr>
<tr>
<td>SquarePyramidal</td>
<td>Square pyramidal tiling (e.g. Shibumi).</td>
</tr>
<tr>
<td>HexagonalPyramidal</td>
<td>Hexagonal pyramidal tiling.</td>
</tr>
<tr>
<td>Concentric</td>
<td>Concentric tiling (e.g. Morris boards, wheel boards, ...).</td>
</tr>
<tr>
<td>Circle</td>
<td>Circular tiling (e.g. Round Merels).</td>
</tr>
<tr>
<td>Spiral</td>
<td>Spiral tiling (e.g. Mehen).</td>
</tr>
<tr>
<td>Dual</td>
<td>Tiling derived from the weak dual of a graph.</td>
</tr>
<tr>
<td>Brick</td>
<td>Brick tiling using 1x2 rectangular brick tiles.</td>
</tr>
<tr>
<td>Mesh</td>
<td>Mesh formed by random spread of points within an outline shape.</td>
</tr>
<tr>
<td>Morris</td>
<td>Morris tiling with concentric square rings and empty centre.</td>
</tr>
<tr>
<td>Celtic</td>
<td>Tiling on a square grid based on Celtic knotwork.</td>
</tr>
<tr>
<td>QuadHex</td>
<td>Quadhex board consisting of a hexagon tessellated by quadrilaterals (e.g. Three Player Chess).</td>
</tr>
</tbody>
</table>

### 16.1.2 hiddenData

Defines possible data to be hidden.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>What</td>
<td>The id of the component on the location is hidden.</td>
</tr>
<tr>
<td>Who</td>
<td>The owner of the component of the location is hidden.</td>
</tr>
<tr>
<td>State</td>
<td>The local state of the location is hidden.</td>
</tr>
<tr>
<td>Count</td>
<td>The number of components on the location is hidden.</td>
</tr>
<tr>
<td>Rotation</td>
<td>The rotation of the component on the location is hidden.</td>
</tr>
<tr>
<td>Value</td>
<td>The piece value of the component on the location is hidden.</td>
</tr>
</tbody>
</table>

### 16.1.3 landmarkType

Defines certain landmarks that can be used to specify individual sites on the board.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CentreSite</td>
<td>The central site of the board.</td>
</tr>
<tr>
<td>LeftSite</td>
<td>The site that is furthest to the left.</td>
</tr>
<tr>
<td>RightSite</td>
<td>The site that is furthest to the right</td>
</tr>
<tr>
<td>Topsite</td>
<td>The site that is furthest to the top.</td>
</tr>
<tr>
<td>BottomSite</td>
<td>The site that is furthest to the bottom.</td>
</tr>
<tr>
<td>FirstSite</td>
<td>The first site indexed in the graph.</td>
</tr>
<tr>
<td>LastSite</td>
<td>The last site indexed in the graph.</td>
</tr>
</tbody>
</table>
16.1.4 puzzleElementType
Defines the possible types of variables that can be used in deduction puzzles.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell</td>
<td>A variable corresponding to a cell.</td>
</tr>
<tr>
<td>Edge</td>
<td>A variable corresponding to an edge.</td>
</tr>
<tr>
<td>Vertex</td>
<td>A variable corresponding to a vertex.</td>
</tr>
<tr>
<td>Hint</td>
<td>A variable corresponding to a hint.</td>
</tr>
</tbody>
</table>

16.1.5 regionTypeDynamic
Defines regions which can change during play.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty</td>
<td>All the empty sites of the current state.</td>
</tr>
<tr>
<td>NotEmpty</td>
<td>All the occupied sites of the current state.</td>
</tr>
<tr>
<td>Own</td>
<td>All the sites occupied by a piece of the mover.</td>
</tr>
<tr>
<td>NotOwn</td>
<td>All the sites not occupied by a piece of the mover.</td>
</tr>
<tr>
<td>Enemy</td>
<td>All the sites occupied by a piece of an enemy of the mover.</td>
</tr>
<tr>
<td>NotEnemy</td>
<td>All the sites empty or occupied by a Neutral piece.</td>
</tr>
</tbody>
</table>

16.1.6 regionTypeStatic
Defines known (predefined) regions of the board.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rows</td>
<td>Row areas.</td>
</tr>
<tr>
<td>Columns</td>
<td>Column areas.</td>
</tr>
<tr>
<td>AllDirections</td>
<td>All direction areas.</td>
</tr>
<tr>
<td>HintRegions</td>
<td>Hint areas.</td>
</tr>
<tr>
<td>Layers</td>
<td>Layers areas.</td>
</tr>
<tr>
<td>Diagonals</td>
<td>diagonal areas.</td>
</tr>
<tr>
<td>SubGrids</td>
<td>SubGrid areas.</td>
</tr>
<tr>
<td>Regions</td>
<td>Region areas.</td>
</tr>
<tr>
<td>Vertices</td>
<td>Vertex areas.</td>
</tr>
<tr>
<td>Corners</td>
<td>Corner areas.</td>
</tr>
<tr>
<td>Sides</td>
<td>Side areas.</td>
</tr>
<tr>
<td>SidesNoCorners</td>
<td>Side areas that are not corners.</td>
</tr>
<tr>
<td>AllSites</td>
<td>All site areas.</td>
</tr>
<tr>
<td>Touching</td>
<td>Touching areas.</td>
</tr>
</tbody>
</table>
16.1.7  relationType

Defines the possible relation types between graph elements.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthogonal</td>
<td>Orthogonal relation.</td>
</tr>
<tr>
<td>Diagonal</td>
<td>Diagonal relation.</td>
</tr>
<tr>
<td>OffDiagonal</td>
<td>Diagonal-off relation.</td>
</tr>
<tr>
<td>Adjacent</td>
<td>Adjacent relation.</td>
</tr>
<tr>
<td>All</td>
<td>Any relation.</td>
</tr>
</tbody>
</table>

16.1.8  shapeType

Defines shape types for known board shapes.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoShape</td>
<td>No defined board shape.</td>
</tr>
<tr>
<td>Custom</td>
<td>Custom board shape defined by the user.</td>
</tr>
<tr>
<td>Square</td>
<td>Square board shape.</td>
</tr>
<tr>
<td>Rectangle</td>
<td>Rectangular board shape.</td>
</tr>
<tr>
<td>Triangle</td>
<td>Triangular board shape.</td>
</tr>
<tr>
<td>Hexagon</td>
<td>Hexagonal board shape.</td>
</tr>
<tr>
<td>Cross</td>
<td>Cross board shape.</td>
</tr>
<tr>
<td>Diamond</td>
<td>Diamond board shape.</td>
</tr>
<tr>
<td>Prism</td>
<td>Diamond board shape extended vertically.</td>
</tr>
<tr>
<td>Quadrilateral</td>
<td>General quadrilateral board shape.</td>
</tr>
<tr>
<td>Rhombus</td>
<td>Rhombus board shape.</td>
</tr>
<tr>
<td>Wheel</td>
<td>Wheel board shape.</td>
</tr>
<tr>
<td>Circle</td>
<td>Circular board shape.</td>
</tr>
<tr>
<td>Spiral</td>
<td>Spiral board shape.</td>
</tr>
<tr>
<td>Wedge</td>
<td>Wedge shape of height N with 1 vertex at the top and 3 vertices on the bottom, for Alquerque boards.</td>
</tr>
<tr>
<td>Star</td>
<td>Multi-pointed star shape.</td>
</tr>
<tr>
<td>Limping</td>
<td>Alternating sides are staggered.</td>
</tr>
<tr>
<td>Regular</td>
<td>Regular polygon with sides of the same length.</td>
</tr>
<tr>
<td>Polygon</td>
<td>General polygon.</td>
</tr>
</tbody>
</table>

16.1.9  siteType

Defines the element types that make up each graph.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>Graph vertex.</td>
</tr>
<tr>
<td>Edge</td>
<td>Graph edge.</td>
</tr>
<tr>
<td>Cell</td>
<td>Graph cell/face.</td>
</tr>
</tbody>
</table>

### 16.1.10 stepType

Defines possible “turtle steps” for describing walks through adjacent sites.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Forward a step.</td>
</tr>
<tr>
<td>L</td>
<td>Turn left a step.</td>
</tr>
<tr>
<td>R</td>
<td>Turn right a step.</td>
</tr>
</tbody>
</table>

### 16.1.11 storeType

Defines the different stores for a mancala board.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No store.</td>
</tr>
<tr>
<td>Outer</td>
<td>Outer store.</td>
</tr>
<tr>
<td>Inner</td>
<td>Inner store.</td>
</tr>
</tbody>
</table>

### 16.1.12 tilingBoardlessType

Defines supported tiling types for boardless games.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>Square tiling.</td>
</tr>
<tr>
<td>Triangular</td>
<td>Triangular tiling.</td>
</tr>
<tr>
<td>Hexagonal</td>
<td>Hexagonal tiling.</td>
</tr>
</tbody>
</table>

### 16.1.13 trackStepType

Defines special steps for describing tracks on the board.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Off the track.</td>
</tr>
<tr>
<td>End</td>
<td>End of the track.</td>
</tr>
<tr>
<td>Repeat</td>
<td>Repeat stepping in the current direction.</td>
</tr>
</tbody>
</table>
16.2 Component

Component types are constant values for specifying various aspects of components in the game. These can include pieces, cards, dice, and so on.

16.2.1 cardType

Defines possible rank values of cards.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joker</td>
<td>Joker rank.</td>
</tr>
<tr>
<td>Ace</td>
<td>Ace rank.</td>
</tr>
<tr>
<td>Two</td>
<td>Two rank.</td>
</tr>
<tr>
<td>Three</td>
<td>Three rank.</td>
</tr>
<tr>
<td>Four</td>
<td>Four rank.</td>
</tr>
<tr>
<td>Five</td>
<td>Five rank.</td>
</tr>
<tr>
<td>Six</td>
<td>Six rank.</td>
</tr>
<tr>
<td>Seven</td>
<td>Seven rank.</td>
</tr>
<tr>
<td>Eight</td>
<td>Eight rank.</td>
</tr>
<tr>
<td>Nine</td>
<td>Nine rank.</td>
</tr>
<tr>
<td>Ten</td>
<td>Ten rank.</td>
</tr>
<tr>
<td>Jack</td>
<td>Jack rank.</td>
</tr>
<tr>
<td>Queen</td>
<td>Queen rank.</td>
</tr>
<tr>
<td>King</td>
<td>King rank.</td>
</tr>
</tbody>
</table>

16.2.2 dealableType

Specifies which types of components can be dealt.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dominoes</td>
<td>Domino component.</td>
</tr>
<tr>
<td>Cards</td>
<td>Card component.</td>
</tr>
</tbody>
</table>

16.2.3 suitType

Defines the possible suit types of cards.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>Club suit.</td>
</tr>
<tr>
<td>Spades</td>
<td>Spade suit.</td>
</tr>
<tr>
<td>Diamonds</td>
<td>Diamond suit.</td>
</tr>
<tr>
<td>Hearts</td>
<td>Heart suit.</td>
</tr>
</tbody>
</table>
16.3 Play

Play types are constant values for specifying various aspects of play. These are typically to do with the “start”, “play” and “end” rules.

16.3.1 modeType

Defines the possible modes of play.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternating</td>
<td>Players alternate making discrete moves.</td>
</tr>
<tr>
<td>Simultaneous</td>
<td>Players move at the same time.</td>
</tr>
<tr>
<td>Simulation</td>
<td>Simulation game</td>
</tr>
</tbody>
</table>

16.3.2 passEndType

Defines the possible types of ending results if all players are passed their turn.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw</td>
<td>The game in a draw.</td>
</tr>
<tr>
<td>NoEnd</td>
<td>The game does not end.</td>
</tr>
</tbody>
</table>

16.3.3 prevType

Defines the possible previous states to refer to.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mover</td>
<td>The state corresponding to the previous move.</td>
</tr>
<tr>
<td>MoverLastTurn</td>
<td>The state corresponding to the previous turn.</td>
</tr>
</tbody>
</table>

16.3.4 repetitionType

Defines the possible types of repetition that can occur in a game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SituationalInTurn</td>
<td>Situational State repeated within a turn.</td>
</tr>
<tr>
<td>PositionalInTurn</td>
<td>Positional State repeated within a turn.</td>
</tr>
<tr>
<td>Positional</td>
<td>State repeated within a game (pieces on the board only).</td>
</tr>
<tr>
<td>Situational</td>
<td>State repeated within a game (all data in the state).</td>
</tr>
</tbody>
</table>
16.3.5 resultType
Defines expected outcomes for each game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Win</td>
<td>Somebody wins.</td>
</tr>
<tr>
<td>Loss</td>
<td>Somebody loses.</td>
</tr>
<tr>
<td>Draw</td>
<td>Nobody wins.</td>
</tr>
<tr>
<td>Tie</td>
<td>Everybody wins.</td>
</tr>
<tr>
<td>Abandon</td>
<td>Game abandoned, typically for being too long.</td>
</tr>
<tr>
<td>Crash</td>
<td>Game stopped due to run-time error.</td>
</tr>
</tbody>
</table>

16.3.6 roleType
Defines the possible role types of the players in a game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutral</td>
<td>Neutral role, owned by nobody.</td>
</tr>
<tr>
<td>P1</td>
<td>Player 1.</td>
</tr>
<tr>
<td>P2</td>
<td>Player 2.</td>
</tr>
<tr>
<td>P3</td>
<td>Player 3.</td>
</tr>
<tr>
<td>P4</td>
<td>Player 4.</td>
</tr>
<tr>
<td>P5</td>
<td>Player 5.</td>
</tr>
<tr>
<td>P6</td>
<td>Player 6.</td>
</tr>
<tr>
<td>P7</td>
<td>Player 7.</td>
</tr>
<tr>
<td>P8</td>
<td>Player 8.</td>
</tr>
<tr>
<td>P9</td>
<td>Player 9.</td>
</tr>
<tr>
<td>P10</td>
<td>Player 10.</td>
</tr>
<tr>
<td>P11</td>
<td>Player 11.</td>
</tr>
<tr>
<td>P12</td>
<td>Player 12.</td>
</tr>
<tr>
<td>P13</td>
<td>Player 13.</td>
</tr>
<tr>
<td>P14</td>
<td>Player 14.</td>
</tr>
<tr>
<td>P15</td>
<td>Player 15.</td>
</tr>
<tr>
<td>P16</td>
<td>Player 16.</td>
</tr>
<tr>
<td>Team1</td>
<td>Team 1 (index 1).</td>
</tr>
<tr>
<td>Team2</td>
<td>Team 2 (index 2).</td>
</tr>
<tr>
<td>Team3</td>
<td>Team 3 (index 3).</td>
</tr>
<tr>
<td>Team4</td>
<td>Team 4 (index 4).</td>
</tr>
<tr>
<td>Team5</td>
<td>Team 5 (index 5).</td>
</tr>
<tr>
<td>Team6</td>
<td>Team 6 (index 6).</td>
</tr>
<tr>
<td>Team7</td>
<td>Team 7 (index 7).</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>Team8</td>
<td>Team 8 (index 8).</td>
</tr>
<tr>
<td>Team9</td>
<td>Team 9 (index 9).</td>
</tr>
<tr>
<td>Team10</td>
<td>Team 10 (index 10).</td>
</tr>
<tr>
<td>Team11</td>
<td>Team 11 (index 11).</td>
</tr>
<tr>
<td>Team12</td>
<td>Team 12 (index 12).</td>
</tr>
<tr>
<td>Team13</td>
<td>Team 13 (index 13).</td>
</tr>
<tr>
<td>Team14</td>
<td>Team 14 (index 14).</td>
</tr>
<tr>
<td>Team15</td>
<td>Team 15 (index 15).</td>
</tr>
<tr>
<td>Team16</td>
<td>Team 16 (index 16).</td>
</tr>
<tr>
<td>Each</td>
<td>Applies to each player (for iteration), e.g. same piece owned by each player</td>
</tr>
<tr>
<td>Shared</td>
<td>Shared role, shared by all players.</td>
</tr>
<tr>
<td>All</td>
<td>All players.</td>
</tr>
<tr>
<td>Mover</td>
<td>Player who is moving.</td>
</tr>
<tr>
<td>Next</td>
<td>Player who is moving next turn.</td>
</tr>
<tr>
<td>Prev</td>
<td>Player who made the previous decision move.</td>
</tr>
<tr>
<td>NonMover</td>
<td>Players who are not moving.</td>
</tr>
<tr>
<td>Enemy</td>
<td>Enemy players.</td>
</tr>
<tr>
<td>Friend</td>
<td>Friend players (Mover + Allies).</td>
</tr>
<tr>
<td>Ally</td>
<td>Ally players.</td>
</tr>
<tr>
<td>Player</td>
<td>Placeholder for iterator over all players, e.g. from end.ForEach</td>
</tr>
</tbody>
</table>

### 16.3.7 whenType

Defines when to perform certain tests or actions within a game.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StartOfMove</td>
<td>Start of a move.</td>
</tr>
<tr>
<td>EndOfMove</td>
<td>End of a move.</td>
</tr>
<tr>
<td>StartOfTurn</td>
<td>Start of a turn.</td>
</tr>
<tr>
<td>EndOfTurn</td>
<td>End of a turn.</td>
</tr>
<tr>
<td>StartOfRound</td>
<td>Start of a round.</td>
</tr>
<tr>
<td>EndOfRound</td>
<td>End of a round.</td>
</tr>
<tr>
<td>StartOfPhase</td>
<td>Start of a phase.</td>
</tr>
<tr>
<td>EndOfPhase</td>
<td>End of a phase.</td>
</tr>
<tr>
<td>StartOfGame</td>
<td>Start of a game.</td>
</tr>
<tr>
<td>EndOfGame</td>
<td>End of a game.</td>
</tr>
<tr>
<td>StartOfMatch</td>
<td>Start of a match.</td>
</tr>
<tr>
<td>Event Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------</td>
</tr>
<tr>
<td>EndOfMatch</td>
<td>End of a match.</td>
</tr>
<tr>
<td>StartOfSession</td>
<td>Start of a session.</td>
</tr>
<tr>
<td>EndOfSession</td>
<td>End of a session.</td>
</tr>
</tbody>
</table>
Part II

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.
17.1 Metadata

The metadata ludeme is a catch-call for all metadata items.

17.1.1 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"))
)
17.2 Info

The info metadata items.

17.2.1 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.")
  }
 ) )
17.3 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.

17.3.1 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")

17.3.2 author

Specifies the author of the game or ruleset.

Format

(author <string>)

where:
  • <string>: The author of the game.

Example

(author "John Doe")

17.3.3 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.

17.3.4 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.

17.3.5 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).

17.3.6 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.")

17.3.7 origin

Specifies the location of the earliest known origin for this game.

Format

(origin <string>)
where:
  • <string>: Earliest known origin for this game.
Example

例

(origin "1953")

17.3.8 publisher
Specifies the publisher of the game.

Format

(publisher <string>)
where:
  • <string>: The publisher of the game.

Example

例

(publisher "Games Inc.")

17.3.9 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

例

(rules
  "Try to make a line of four."
)
17.3.10  **source**
Specifies the reference for the game, or its currently chosen ruleset.

**Format**

```
(sources <string>)
```

where:

- `<string>`: The source of the game’s rules.

**Example**

```
(sources "Murray 1969")
```

---

17.3.11  **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".
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.
18.1 Board

The (board ...) ‘super’ metadata ludeme is used modify a graphic property of a board.

18.1.1 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 replaceComponentsWithFilledCells:<boolean>)

where:

- replaceComponentsWithFilledCells:<boolean>: True if cells should be filled instead of component drawn.

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 <boardBooleanType> [boolean])

where:

- <boardBooleanType>: The type of data.
- [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>] [scaleX:<float>] [scaleY:<float>] [rotation:int] [offsetX:<float>] [offsetY:<float>])

where:

- <pieceGroundType>: The type of data to apply to the board.
- [image:<string>]: Name of the image to draw. Default value is an outline around the board.
- [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].
- [scaleX:<float>]: Scale for the drawn image, relative to the cell size of the container, along x-axis [1.0].
- [scaleY:<float>]: Scale for the drawn image, relative to the cell size of the container, along y-axis [1.0].
- [rotation:int]: Rotation of the drawn image (clockwise).
- [offsetX:<float>]: Offset distance as percentage of board size to push the image to the right [0].
- [offsetY:<float>]: Offset distance as percentage of board size to push the image to the down [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 placement of the board.

(board Placement [scale:<float>] [offsetX:<float>] [offsetY:<float>])

where:

- [scale:<float>]: scale for the board.
• \([\text{offsetX:<float>}]\): X offset for board center.
• \([\text{offsetY:<float>}]\): Y offset for board center.

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 Placement scale:0.8)

18.1.2 boardBooleanType
Defines the types of Board metadata depending only of a boolean.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checkered</td>
<td>To indicate whether the board should be drawn in a checkered pattern.</td>
</tr>
<tr>
<td>RedrawAfterMove</td>
<td>To indicate whether the board should be redrawn after every move.</td>
</tr>
</tbody>
</table>

18.1.3 boardColourType
Defines the types of Board metadata related to the colour.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colour</td>
<td>To set the colour of a specific aspect of the board.</td>
</tr>
</tbody>
</table>

18.1.4 boardPlacementType
Defines the types of Board metadata related to the placement.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
Placement | To set the placement of the board.

### 18.1.5 boardStylePenAndPaperType

Supported style types for rendering particular boards for specific case of PenAndPaper.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PenAndPaper</td>
<td>General style for the pen &amp; paper style games, such as graph games.</td>
</tr>
</tbody>
</table>

### 18.1.6 boardStyleThicknessType

Defines the types of Board metadata related to the thickness style.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StyleThickness</td>
<td>To set the preferred scale for the thickness of a specific aspect of the board.</td>
</tr>
</tbody>
</table>

### 18.1.7 boardStyleType

Defines the types of Board metadata related to the style.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style</td>
<td>To set the style of the board.</td>
</tr>
</tbody>
</table>
18.2 Hand

The (hand ...) ‘super’ metadata ludeme is used modify a graphic property of a hand.

18.2.1 hand

Sets a graphic data to the hand.

Format

(hand Placement <roleType> [scale:<float>] [offsetX:<float>] [offsetY:<float>] [vertical:<boolean>])

where:

- <roleType>: Roletype owner of the hand.
- [scale:<float>]: Scale for the board.
- [offsetX:<float>]: Offset distance percentage to push the board to the right.
- [offsetY:<float>]: Offset distance percentage to push the board down.
- [vertical:<boolean>]: If the hand should be drawn vertically.

Example

(hand Placement P1 scale:1.0 offsetX:0.5 offsetY:0.5 vertical:True)

18.2.2 handPlacementType

Defines the types of Board metadata related to the placement.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Placement</td>
<td>To set the placement of the hand.</td>
</tr>
</tbody>
</table>
18.3 No

The `(no ...) 'super' metadata ludeme is used to not show a graphic property.

18.3.1 no

Hides a graphic element.

Format

\((\text{no } <\text{noBooleanType}> \ [<\text{boolean}>])\)

where:
- \(<\text{noBooleanType}>\): The type of data.
- \([<\text{boolean}>]\): True if the graphic data has to be hidden [True].

Examples

\((\text{no Board})\)
\((\text{no Animation})\)
\((\text{no HandScale})\)
\((\text{no Curves})\)

18.3.2 noBooleanType

Defines the types of Hide metadata depending only of a boolean.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board</td>
<td>To indicate whether the board should be hidden.</td>
</tr>
<tr>
<td>Animation</td>
<td>To indicate whether the animations should be hidden.</td>
</tr>
<tr>
<td>Sunken</td>
<td>To indicate whether the sunken outline should be drawn.</td>
</tr>
<tr>
<td>HandScale</td>
<td>To indicate whether pieces drawn in the hand should be scaled or not.</td>
</tr>
<tr>
<td>Curves</td>
<td>To indicate if the lines that make up the board’s rings should be drawn as straight lines.</td>
</tr>
<tr>
<td>MaskedColour</td>
<td>To indicate if the colour of the masked players should not be the colour of the player.</td>
</tr>
<tr>
<td>DicePips</td>
<td>To indicate if pips on the dice should be always drawn as a single number.</td>
</tr>
</tbody>
</table>
18.4 Others

The “other” metadata items are used to modify the UI for a given game for more specific data.

18.4.1 autoPass

Indicates if the game should auto-pass when this is the only legal move.

Format

\[(\text{autoPass} \ <\text{boolean}>)]

where:

- \(<\text{boolean}>\): If the game should auto-pass when this is the only legal move [True].

Example

\[(\text{autoPass False})\]

18.4.2 hiddenImage

Draws a specified image when a piece is hidden.

Format

\[(\text{hiddenImage} \ <\text{string}>)]

where:

- \(<\text{string}>\): Name of the hidden Image image to draw.

Example

\[(\text{hiddenImage} \ "door")\]

18.4.3 stackType

Sets the stack design for a container.
Format

(stackType [roleType] [string] [int] [siteType] ([sites:{int}] | [site:int]) [state:int] <pieceStackType> [<float>] [limit:int])

where:

• [roleType]: Player whose index we want to match.
• [string]: Container name to match.
• [int]: Container index to match.
• [siteType]: The GraphElementType for the specified sites [Cell].
• [sites:{int}]: Draw image on all specified sites.
• [site:int]: Draw image on this site.
• [state:int]: Local state to match.
• <pieceStackType>: Stack type for this piece.
• [<float>]: Scaling factor [1.0].
• [limit:int]: Stack limit [5].

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.

18.4.4 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.
18.5 Piece

The (piece ...) ‘super’ metadata ludeme is used to modify a graphic property of a piece.

18.5.1 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 foreground image of a piece.

(piece <pieceGroundType> [role: <roleType>] [string: <string>] [state: int] [value: int] image: <string> [fillColour: <colour>] [edgeColour: <colour>] [scale: <float>] [rotation: int])

where:

- <pieceGroundType>: The type of data to apply to the pieces.
- [role: <roleType>]: Player whose index is to be matched.
- [string]: Base piece name to match.
- [state: int]: State to match.
- [value: int]: Value 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].
- [rotation: int]: Amount of rotation for drawn image.

For setting the colour of a piece.

(piece Colour [role: <roleType>] [string: <string>] [state: int] [value: int] [fillColour: <colour>] [strokeColour: <colour>] [secondaryColour: <colour>])

where:

- [role: <roleType>]: Player whose index is to be matched.
- [string]: Base piece name to match.
- [state: int]: State to match.
- [value: int]: Value to match.
- [fillColour: <colour>]: Fill colour for this piece.
- [strokeColour: <colour>]: Stroke colour for this piece.
- [secondaryColour: <colour>]: Secondary colour for this piece.

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>)

where:
- [<roleType>]: Player whose index is to be matched.
- [<string>]: Base piece name to match.
- <float>: Scaling factor in both x and y direction.

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 Rotate P2 degrees:90)
(piece Scale "Pawn" .5)
(piece Scale "Disc" .5)

18.5.2 pieceColourType
Defines the types of Piece metadata to set a colour.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colour</td>
<td>To set the colour of a piece.</td>
</tr>
</tbody>
</table>

18.5.3 pieceFamiliesType
Defines the types of Piece metadata belonging to some families.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Families</td>
<td>To specify a list of families for the game’s pieces.</td>
</tr>
</tbody>
</table>

18.5.4 pieceGroundType
Defines the types of Piece metadata to set the foreground.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background</td>
<td>To draw a specified image in front of a piece.</td>
</tr>
</tbody>
</table>
Foreground | To draw a specified image behind a piece.
Hidden | To draw a specified image as the hidden symbol.

### 18.5.5 pieceNameType

Defines the types of Piece metadata to change the name.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename</td>
<td>To replace a piece’s name with an alternative.</td>
</tr>
<tr>
<td>ExtendName</td>
<td>To add additional text to a piece name.</td>
</tr>
<tr>
<td>AddStateToName</td>
<td>To add the local state value of a piece to its name.</td>
</tr>
<tr>
<td>Hidden</td>
<td>To set the hidden image for a piece.</td>
</tr>
</tbody>
</table>

### 18.5.6 pieceReflectType

Defines the types of Piece metadata to reflect.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflect</td>
<td>To indicate whether to apply any vertical or horizontal image reflections to a piece.</td>
</tr>
</tbody>
</table>

### 18.5.7 pieceRotateType

Defines the types of Piece metadata to reflect.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotate</td>
<td>To indicate whether to rotate the image for a piece.</td>
</tr>
</tbody>
</table>

### 18.5.8 pieceScaleByType

Defines the types of Piece metadata to scale by a data.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByValue</td>
<td>To indicate if the pieces in the game should be scaled in size based on their value.</td>
</tr>
</tbody>
</table>

### 18.5.9 pieceScaleType

Defines the types of Piece metadata to scale.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale</td>
<td>To set the image scale of a piece.</td>
</tr>
</tbody>
</table>
18.5.10 pieceStyleType

Defines the types of Piece metadata to set a style.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Style</td>
<td>To set the style of a piece.</td>
</tr>
</tbody>
</table>
18.6 Player

The “player” metadata items describe relevant player settings.

18.6.1 player

Sets a graphic element to a player.

**Format**

For setting the colour of a player.

(player Colour <roleType> <colour>)

where:

- <roleType>: Player whose index is to be matched.
- <colour>: Colour wanted for this player.

For setting the name of a player.

(player Name <roleType> <string>)

where:

- <roleType>: Player whose index is to be matched.
- <string>: Name wanted for this player.

**Examples**

(player Colour P1 (colour Black))

(player Name P1 "Player 1")

18.6.2 playerColourType

Defines the types of Player metadata depending of a colour.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colour</td>
<td>To set the colour of a player.</td>
</tr>
</tbody>
</table>

18.6.3 playerNameType

Defines the types of Player metadata depending of a name.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>To set the name of a player.</td>
</tr>
</tbody>
</table>
18.7 Puzzle

The “Puzzle” metadata items describe relevant puzzle settings.

18.7.1 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.

18.7.2 drawHint

Indicates how the hints for the puzzle should be shown.

Format

(drawHint <puzzleDrawHintType>)

where:

• <puzzleDrawHintType>: How hints should be shown.

Example

(drawHint TopLeft)
18.7.3 hintLocation
Indicates how to determine the site for the hint to be drawn.

Format

```
(hintLocation <puzzleHintLocationType>)
```

where:
- `<puzzleHintLocationType>`: How to determine hint location.

Example

```
(hintLocation BetweenVertices)
```
18.8 Region

The (region ...) ‘super’ metadata ludeme is used to modify a graphic property of a region.

18.8.1 region

Sets a graphic element to a region.

Format

\[
\text{format} \quad \text{(region Colour [<string>] [<roleType>] [<siteType>] ([{int}] | [int]) [region] [regionSiteType:<siteType>] [colour] [scale:<float>])}
\]

where:
- [<string>]: Region to be coloured.
- [<roleType>]: Player whose index is to be matched (only for Region).
- [<siteType>]: The GraphElementType for the specified sites [DefaultBoardType].
- [{int}]: Sites to be coloured.
- [int]: Site to be coloured.
- [region]: RegionFunction to be coloured.
- [regionSiteType:<siteType>]: The SiteType of the region [DefaultBoardType].
- [colour]: The assigned colour for the specified boardGraphicsType.
- [scale:<float>]: The scale for the region graphics (only applies to Edge siteType).

Example

\[
\text{(region Colour "Home" Edge regionSiteType:Cell (colour Black))}
\]

18.8.2 regionColourType

Defines the types of Region metadata depending of a colour.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colour</td>
<td>To set the colour of a player.</td>
</tr>
</tbody>
</table>
18.9 Show

The (show ...) ‘super’ metadata ludeme is used to show a graphic property or an information during the game.

18.9.1 show

Shows a graphic property or an information.

Format

For showing properties on the holes.

(show AsHoles {int} <holeType>)

where:

- {int}: The list of indices of the holes.
- <holeType>: The shape of the holes.

For showing symbols on sites.

(show Symbol <string> [string] [roleType] [siteType] ([{int}] | [int]) [region] [boardGraphicsType] [fillColour:colour] [edgeColour:colour] [scale:float] [rotation:int] [offsetX:float] [offsetY:float])

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.
- [region]: Draw image on this regionFunction.
- [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.
• [offsetX:<float>]: Horizontal offset for image (to the right) [0.0].
• [offsetY:<float>]: Vertical offset for image (downwards) [0.0].

For showing symbols on sites.

(show Line {{int}} [<siteType>] [<colour>] [scale:<float>] [curve:{<float>}] [<curveType>])

where:
• {{int}}: The line to draw (pairs of vertices).
• [<siteType>]: The GraphElementType for the specified sites [Vertex].
• [<colour>]: The colour of the line.
• [scale:<float>]: The scale of the line.
• [curve:{<float>}] for creating a Bézier curve (4 values: x1, y1, x2, y2, between 0 and 1).
• [<curveType>]: Type of curve [Spline].

For showing specific edges of the graph board (only valid with GraphStyle or its children).

(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] [offsetImage: boolean]
   [valueOutline: boolean])

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].
- `[offsetImage: boolean]`: Offset the image by the size of the displayed value [False].
- `[valueOutline: boolean]`: Draw outline around the displayed value [False].

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] [roleType] [int] [string])

where:
- `[whenScoreType]`: When the score should be shown [Always].
- `[roleType]`: Player whose index is to be matched [All].
- `[int]`: Replacement value to display instead of score.
- `[string]`: Extra string to append to the score displayed [" "].

Examples
(show AsHoles {5 10} Square)
(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)

18.9.2 showBooleanType
Defines the types of Show metadata depending only of a boolean.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pits</td>
<td>To indicates whether the pits on the board should be marked with their owner.</td>
</tr>
<tr>
<td>PlayerHoles</td>
<td>To indicates whether the player’s holes on the board should be marked with their owner.</td>
</tr>
<tr>
<td>LocalStateHoles</td>
<td>To indicates whether the holes with a local state of zero should be marked.</td>
</tr>
<tr>
<td>RegionOwner</td>
<td>To indicates whether the owner of each region should be shown.</td>
</tr>
<tr>
<td>Cost</td>
<td>To indicates whether the cost of the graph element has to be shown.</td>
</tr>
<tr>
<td>Hints</td>
<td>To indicates whether the hints of the puzzle has to be shown.</td>
</tr>
<tr>
<td>EdgeDirections</td>
<td>To indicates whether the edge directions should be shown.</td>
</tr>
<tr>
<td>PossibleMoves</td>
<td>To indicates whether the possible moves are always shown.</td>
</tr>
<tr>
<td>CurvedEdges</td>
<td>To indicates whether curved edges should be shown.</td>
</tr>
<tr>
<td>StraightEdges</td>
<td>To indicates whether straight edges should be shown.</td>
</tr>
</tbody>
</table>

18.9.3 showCheckType
Defines the types of Show metadata for a check.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>To indicate whether a &quot;Check&quot; should be displayed when a piece is in threatened.</td>
</tr>
</tbody>
</table>

### 18.9.4 showComponentDataType

Defines the types of data to show for a component in the super metadata Show ludeme.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>To indicate whether the state of a piece should be displayed.</td>
</tr>
<tr>
<td>Value</td>
<td>To indicate whether the value of a piece should be displayed.</td>
</tr>
</tbody>
</table>

### 18.9.5 showComponentType

Defines the types of Show metadata for a type of components.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piece</td>
<td>To indicate that the component to apply the type is a piece.</td>
</tr>
</tbody>
</table>

### 18.9.6 showEdgeType

Defines the types of Draw metadata related to edges.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edges</td>
<td>To specify customised drawing of edges in the board graph.</td>
</tr>
</tbody>
</table>

### 18.9.7 showLineType

Defines the types of Draw metadata related to line.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line</td>
<td>To draw a specified image on the board.</td>
</tr>
</tbody>
</table>

### 18.9.8 showScoreType

Defines the types of Show metadata for a score.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score</td>
<td>To indicate whether the score should be shown only in certain situations.</td>
</tr>
</tbody>
</table>

### 18.9.9 showSiteDataType

Defines the types of Show metadata related to a data of the sites.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AsHoles</td>
<td>To indicate whether the sites of the board should be represented as holes.</td>
</tr>
</tbody>
</table>

### 18.9.10 showSiteType

Defines the types of Show metadata related to a site.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sites</td>
<td>To apply it on the sites.</td>
</tr>
<tr>
<td>Cell</td>
<td>To apply it on the cells.</td>
</tr>
</tbody>
</table>

### 18.9.11 showSymbolType

Defines the types of Draw metadata related to symbol.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol</td>
<td>To draw a specified image on the board.</td>
</tr>
</tbody>
</table>

### 18.10 Util

The “util” metadata items are used for setting miscellaneous properties of the current game.

#### 18.10.1 boardGraphicsType

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InnerEdges</td>
<td>Edges that are not along a board side.</td>
</tr>
<tr>
<td>OuterEdges</td>
<td>Edges that define a board side.</td>
</tr>
<tr>
<td>Phase0</td>
<td>Cells in phase 0, e.g. dark cells on the Chess board.</td>
</tr>
<tr>
<td>Phase1</td>
<td>Cells in phase 1, e.g. light cells on the Chess board.</td>
</tr>
<tr>
<td>Phase2</td>
<td>Cells in phase 2, e.g. for hexagonal tiling colouring.</td>
</tr>
<tr>
<td>Phase3</td>
<td>Cells in phase 3, e.g. for exotic tiling colourings.</td>
</tr>
<tr>
<td>Phase4</td>
<td>Cells in phase 4, e.g. for exotic tiling colouring.</td>
</tr>
<tr>
<td>Phase5</td>
<td>Cells in phase 5, e.g. for exotic tiling colourings.</td>
</tr>
<tr>
<td>Symbols</td>
<td>Symbols drawn on the board, e.g. Senet, Hnefatafl, Royal Game of Ur...</td>
</tr>
<tr>
<td>InnerVertices</td>
<td>Intersections of lines on the board, e.g. where Go stones are played.</td>
</tr>
<tr>
<td>OuterVertices</td>
<td>Intersections of lines on the board, along the perimeter of the board.</td>
</tr>
</tbody>
</table>
### 18.10.2 componentStyleType

Supported style types for rendering particular components.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Piece</td>
<td>Style for pieces.</td>
</tr>
<tr>
<td>Tile</td>
<td>Style for tiles (components that fill a cell and may have marked paths).</td>
</tr>
<tr>
<td>Card</td>
<td>Style for playing cards.</td>
</tr>
<tr>
<td>Die</td>
<td>Style for die components used as playing pieces.</td>
</tr>
<tr>
<td>Domino</td>
<td>Style for dominoes</td>
</tr>
<tr>
<td>LargePiece</td>
<td>Style for large pieces that straddle more than once site, e.g. the L Game.</td>
</tr>
<tr>
<td>ExtendedShogi</td>
<td>Extended style for Shogi pieces.</td>
</tr>
<tr>
<td>ExtendedXiangqi</td>
<td>Extended style for Shogi pieces.</td>
</tr>
<tr>
<td>NativeAmericanDice</td>
<td>Style for native american dice.</td>
</tr>
</tbody>
</table>

### 18.10.3 containerStyleType

Supported style types for rendering particular boards.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Board</td>
<td>General style for boards.</td>
</tr>
<tr>
<td>Hand</td>
<td>General style for player hands.</td>
</tr>
<tr>
<td>Deck</td>
<td>General style for player Decks.</td>
</tr>
<tr>
<td>Dice</td>
<td>General style for player Dice.</td>
</tr>
<tr>
<td>Boardless</td>
<td>General style for boardless games, e.g. Andantino.</td>
</tr>
<tr>
<td>ConnectiveGoal</td>
<td>General board style for games with connective goals.</td>
</tr>
<tr>
<td>Mancala</td>
<td>General style for Mancala boards.</td>
</tr>
<tr>
<td>PenAndPaper</td>
<td>General style for the pen &amp; paper style games, such as graph games.</td>
</tr>
<tr>
<td>Shibumi</td>
<td>Style for square pyramidal games played on the Shibumi board, e.g. Spline.</td>
</tr>
<tr>
<td>Spiral</td>
<td>General style for games played on a spiral board, e.g. Mehen.</td>
</tr>
<tr>
<td>Isometric</td>
<td>General style for games played on an isometric board.</td>
</tr>
<tr>
<td>Puzzle</td>
<td>General style for deduction puzzle boards.</td>
</tr>
<tr>
<td>Agon</td>
<td>Custom style for the Agon board.</td>
</tr>
<tr>
<td>Backgammon</td>
<td>Custom style for the Backgammon board.</td>
</tr>
<tr>
<td>Chess</td>
<td>Custom style for the Chess board.</td>
</tr>
<tr>
<td>Connect4</td>
<td>Custom style for the Connect4 board.</td>
</tr>
<tr>
<td>Go</td>
<td>Custom style for the Go board.</td>
</tr>
<tr>
<td>Graph</td>
<td>General style for graph game boards.</td>
</tr>
<tr>
<td><strong>HoundsAndJackals</strong></td>
<td>Custom style for the Hounds and Jackals (58 Holes) board.</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Janggi</strong></td>
<td>Custom style for the Janggi board.</td>
</tr>
<tr>
<td><strong>Lasca</strong></td>
<td>Custom style for the Lasca board.</td>
</tr>
<tr>
<td><strong>Shogi</strong></td>
<td>Custom style for the Shogi board.</td>
</tr>
<tr>
<td><strong>SnakesAndLadders</strong></td>
<td>Custom style for the Snakes and Ladders board.</td>
</tr>
<tr>
<td><strong>Surakarta</strong></td>
<td>Custom style for the Surakarta board.</td>
</tr>
<tr>
<td><strong>Table</strong></td>
<td>Custom style for the Table board.</td>
</tr>
<tr>
<td><strong>Tafl</strong></td>
<td>Custom style for Tafl boards.</td>
</tr>
<tr>
<td><strong>Xiangqi</strong></td>
<td>Custom style for the Xiangqi board.</td>
</tr>
<tr>
<td><strong>UltimateTicTacToe</strong></td>
<td>Custom style for the Ultimate Tic-Tac-Toe board.</td>
</tr>
<tr>
<td><strong>Futoshiki</strong></td>
<td>Custom style for the Futoshiki puzzle board.</td>
</tr>
<tr>
<td><strong>Hashi</strong></td>
<td>Custom style for the Hashi puzzle board.</td>
</tr>
<tr>
<td><strong>Kakuro</strong></td>
<td>Custom style for the Kakuro puzzle board.</td>
</tr>
<tr>
<td><strong>Sudoku</strong></td>
<td>Custom style for the Sudoku board.</td>
</tr>
</tbody>
</table>

**18.10.4 controllerType**

Defines supported controller types for handling user interactions for particular topologies.

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicController</td>
<td>Basic user interaction controller.</td>
</tr>
<tr>
<td>PyramidalController</td>
<td>User interaction controller for games played on pyramidal topologies.</td>
</tr>
</tbody>
</table>

**18.10.5 curveType**

Supported style types for drawing curves.

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Spline</td>
<td>Spline curve based on relative distances.</td>
</tr>
<tr>
<td>Bezier</td>
<td>Bezier curve based on absolute distances.</td>
</tr>
</tbody>
</table>

**18.10.6 edgeType**

Defines edge type for drawing board elements, e.g. for graph games.

<table>
<thead>
<tr>
<th><strong>Value</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All board edges.</td>
</tr>
<tr>
<td>Inner</td>
<td>Inner board edges.</td>
</tr>
<tr>
<td>Outer</td>
<td>Outer board edges.</td>
</tr>
<tr>
<td>Interlayer</td>
<td>Interlayer board edges.</td>
</tr>
</tbody>
</table>
18.10.7 holeType
Defines hole styles for Mancala board.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Square</td>
<td>Hole as Square.</td>
</tr>
<tr>
<td>Oval</td>
<td>Oval as Square.</td>
</tr>
</tbody>
</table>

18.10.8 lineStyle
Defines line styles for drawing board elements, e.g. edges for graph games.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thin</td>
<td>Thin line.</td>
</tr>
<tr>
<td>Thick</td>
<td>Thick line.</td>
</tr>
<tr>
<td>ThinDotted</td>
<td>Thin dotted line.</td>
</tr>
<tr>
<td>ThickDotted</td>
<td>Thick dotted line.</td>
</tr>
<tr>
<td>ThinDashed</td>
<td>Thin dashed line.</td>
</tr>
<tr>
<td>ThickDashed</td>
<td>Thick dashed line.</td>
</tr>
<tr>
<td>Hidden</td>
<td>Line not drawn.</td>
</tr>
</tbody>
</table>

18.10.9 pieceStackType
Defines different ways of visualising stacks of pieces.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Stacked one above the other (with offset).</td>
</tr>
<tr>
<td>Ground</td>
<td>Spread on the ground, e.g. Snakes and Ladders or Pachisi.</td>
</tr>
<tr>
<td>Reverse</td>
<td>Reverse stacking downwards.</td>
</tr>
<tr>
<td>Fan</td>
<td>Spread to show each component like a hand of cards.</td>
</tr>
<tr>
<td>FanAlternating</td>
<td>Spread to show each component like a hand of cards, alternating left and right side of centre.</td>
</tr>
<tr>
<td>None</td>
<td>No visible stacking.</td>
</tr>
<tr>
<td>Backgammon</td>
<td>Stacked Backgammon-style in lines of five.</td>
</tr>
<tr>
<td>Count</td>
<td>Show just top piece, with the stack value as number.</td>
</tr>
<tr>
<td>CountColoured</td>
<td>Show just top piece, with the stack value as number(s), coloured by who.</td>
</tr>
<tr>
<td>Ring</td>
<td>Stacked Ring-style around cell perimeter.</td>
</tr>
<tr>
<td>TowardsCenter</td>
<td>Stacked towards the center of the board.</td>
</tr>
</tbody>
</table>

18.10.10 puzzleDrawHintType
Defines different ways of visualising stacks of pieces.
<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Hints drawn in the middle.</td>
</tr>
<tr>
<td>TopLeft</td>
<td>Hints drawn in the top left.</td>
</tr>
<tr>
<td>NextTo</td>
<td>Draw the hint next to the region.</td>
</tr>
<tr>
<td>None</td>
<td>No hints.</td>
</tr>
</tbody>
</table>

**18.10.11 puzzleHintLocationType**

Defines different ways of visualising stacks of pieces.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>Hints placed on top-left site of region.</td>
</tr>
<tr>
<td>BetweenVertices</td>
<td>Draw hint on edge between vertex.</td>
</tr>
</tbody>
</table>

**18.10.12 valueLocationType**

Specified where to draw state of an item in the interface, relative to its position.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>No location.</td>
</tr>
<tr>
<td>Corner</td>
<td>At the top left corner of the item’s location.</td>
</tr>
<tr>
<td>Top</td>
<td>At the top of the item’s location.</td>
</tr>
<tr>
<td>Middle</td>
<td>Centred on the item’s location.</td>
</tr>
</tbody>
</table>

**18.10.13 whenScoreType**

Specifies when to show player scores to the user.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always</td>
<td>Always show player scores.</td>
</tr>
<tr>
<td>Never</td>
<td>Never show player scores.</td>
</tr>
<tr>
<td>AtEnd</td>
<td>Only show player scores at end of game.</td>
</tr>
</tbody>
</table>
18.11 Util - Colour

The “colour” metadata items allow the user to specify preferred colours for use in other metadata items.

18.11.1 colour

Defines a colour for use in metadata items.

**Format**

For defining a colour with the Red Green Blue values.

\[(\text{colour } \text{int int int})\]

where:

- \text{int}: Red component \([0..255]\).
- \text{int}: Green component \([0..255]\).
- \text{int}: Blue component \([0..255]\).

For defining a colour with the Red Green Blue values and an alpha value.

\[(\text{colour } \text{int int int int})\]

where:

- \text{int}: Red component \([0..255]\).
- \text{int}: Green component \([0..255]\).
- \text{int}: Blue component \([0..255]\).
- \text{int}: Alpha component \([0..255]\).

For defining a colour with the hex code.

\[(\text{colour } \text{<string>})\]

where:

- \text{<string>}: Six digit hexadecimal code.

For defining a predefined colour.

\[(\text{colour } \text{<userColourType>})\]
where:

- `<userColourType>`: Predefined user colour type.

### Examples

(colour 255 0 0)
(colour 255 0 0 127)
(colour "#00ff1a")
(colour DarkBlue)

### 18.11.2 userColourType

Specifies the colour of the user.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White</td>
<td>Plain white.</td>
</tr>
<tr>
<td>Black</td>
<td>Plain black.</td>
</tr>
<tr>
<td>Grey</td>
<td>Medium grey.</td>
</tr>
<tr>
<td>LightGrey</td>
<td>Light grey.</td>
</tr>
<tr>
<td>VeryLightGrey</td>
<td>Very light grey.</td>
</tr>
<tr>
<td>DarkGrey</td>
<td>Dark grey.</td>
</tr>
<tr>
<td>VeryDarkGrey</td>
<td>Very dark grey.</td>
</tr>
<tr>
<td>Dark</td>
<td>Almost black.</td>
</tr>
<tr>
<td>Red</td>
<td>Plain red.</td>
</tr>
<tr>
<td>Green</td>
<td>Plain green.</td>
</tr>
<tr>
<td>Blue</td>
<td>Blue.</td>
</tr>
<tr>
<td>Yellow</td>
<td>Yellow.</td>
</tr>
<tr>
<td>Pink</td>
<td>Pink.</td>
</tr>
<tr>
<td>Cyan</td>
<td>Cyan.</td>
</tr>
<tr>
<td>Brown</td>
<td>Medium brown.</td>
</tr>
<tr>
<td>DarkBrown</td>
<td>Dark brown.</td>
</tr>
<tr>
<td>VeryDarkBrown</td>
<td>Very dark brown.</td>
</tr>
<tr>
<td>Purple</td>
<td>Purple.</td>
</tr>
<tr>
<td>Magenta</td>
<td>Magenta.</td>
</tr>
<tr>
<td>Turquoise</td>
<td>Turquoise.</td>
</tr>
<tr>
<td>Orange</td>
<td>Orange.</td>
</tr>
<tr>
<td>LightOrange</td>
<td>Light orange.</td>
</tr>
<tr>
<td>Colour</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>LightRed</td>
<td>Light red.</td>
</tr>
<tr>
<td>DarkRed</td>
<td>Dark red.</td>
</tr>
<tr>
<td>Burgundy</td>
<td>Burgundy.</td>
</tr>
<tr>
<td>LightGreen</td>
<td>Light green.</td>
</tr>
<tr>
<td>DarkGreen</td>
<td>Dark green.</td>
</tr>
<tr>
<td>LightBlue</td>
<td>Light blue.</td>
</tr>
<tr>
<td>VeryLightBlue</td>
<td>Very light blue.</td>
</tr>
<tr>
<td>DarkBlue</td>
<td>Dark blue.</td>
</tr>
<tr>
<td>IceBlue</td>
<td>Light icy blue.</td>
</tr>
<tr>
<td>Gold</td>
<td>Gold.</td>
</tr>
<tr>
<td>Silver</td>
<td>Silver.</td>
</tr>
<tr>
<td>Bronze</td>
<td>Bronze.</td>
</tr>
<tr>
<td>GunMetal</td>
<td>Gun metal blue.</td>
</tr>
<tr>
<td>HumanLight</td>
<td>Light human skin tone.</td>
</tr>
<tr>
<td>HumanDark</td>
<td>Dark human skin tone.</td>
</tr>
<tr>
<td>Cream</td>
<td>Cream.</td>
</tr>
<tr>
<td>DeepPurple</td>
<td>Deep purple.</td>
</tr>
<tr>
<td>PinkFloyd</td>
<td>Pink.</td>
</tr>
<tr>
<td>BlackSabbath</td>
<td>Very dark bluish black.</td>
</tr>
<tr>
<td>KingCrimson</td>
<td>King of the crimsons.</td>
</tr>
<tr>
<td>TangerineDream</td>
<td>Tangerine.</td>
</tr>
<tr>
<td>BabyBlue</td>
<td>Baby Blue.</td>
</tr>
<tr>
<td>LightTan</td>
<td>Light tan (as per Taff boards).</td>
</tr>
</tbody>
</table>
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.
19.1   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.

19.1.1   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.
19.2 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.

19.2.1 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
19.2.2 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.
19.3 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.

19.3.1 heuristics

Defines a collection of heuristics, which can be used by Alpha-Beta agent in Ludii for their heuristics state evaluations.

Format

For a single heuristic term.

(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) })
19.4 Heuristics - Terms

The terms package includes the actual heuristic metrics that can be applied and their settings.

19.4.1 centreProximity

Defines a heuristic term based on the proximity of pieces to the centre of a game’s board.

Format

\[
\text{(centreProximity [transformation:<heuristicTransformation>] [weight:<float>] [pieceWeights:{<pair>}])}
\]

where:

- \[\text{transformation:<heuristicTransformation>}\]: An optional transformation to be applied to any raw heuristic score outputs.
- \[\text{weight:<float>}\]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
- \[\text{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

\[
\text{(centreProximity pieceWeights:{ (pair "Queen" 1.0) (pair "King" -1.0) })}
\]

19.4.2 componentValues

Defines a heuristic term based on the values of sites that contain components owned by a player.

Format

\[
\text{(componentValues [transformation:<heuristicTransformation>] [weight:<float>] [pieceWeights:{<pair>}] [boardOnly:<boolean>])}
\]

where:

- \[\text{transformation:<heuristicTransformation>}\]: An optional transformation to be applied to any raw heuristic score outputs.
- \[\text{weight:<float>}\]: The weight for this term in a linear combination of multiple terms. If not specified, a default weight of 1.0 is used.
• \texttt{[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. These weights are multiplied with the values on sites where these pieces are present.

• \texttt{[boardOnly:boolean]}

If true, only pieces that are on the game’s main board are counted, and pieces that are, for instance, in players’ hands are excluded. False by default.

Example

\begin{verbatim}
(componentValues boardOnly:True)
\end{verbatim}

19.4.3 cornerProximity

Defines a heuristic term based on the proximity of pieces to the corners of a game’s board.

Format

\begin{verbatim}
(cornerProximity [transformation:<heuristicTransformation>] [weight:<float>] [pieceWeights:{pair}])
\end{verbatim}

where:

• \texttt{[transformation:<heuristicTransformation>]}

An optional transformation to be applied to any raw heuristic score outputs.

• \texttt{[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.

• \texttt{[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

\begin{verbatim}
(cornerProximity pieceWeights:{ (pair "Queen" -1.0) (pair "King" 1.0) })
\end{verbatim}

19.4.4 currentMoverHeuristic

Defines a heuristic term that adds its weight only for the player whose turn it is in any given game state.
19.4.5 influence

Defines a heuristic term that multiplies its weight by the number of moves with distinct "to" positions that a player has in a current game state, divided by the number of playable positions that exist in the game.

Format

(influence [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

(influence weight:0.5)

Remarks

Always produces a score of 0 for players who are not the current mover.
19.4.6 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)

19.4.7 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>}] [boardOnly:<boolean>])

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.
• [boardOnly:<boolean>]: If true, only pieces that are on the game’s main board are counted, and pieces that are, for instance, in players’ hands are excluded. False by default.

Example

(material pieceWeights:{ (pair "Pawn" 1.0) (pair "Bishop" 3.0) })

19.4.8 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.

19.4.9 nullHeuristic

Defines a null heuristic term that always returns a value of 0.

Format

(nullHeuristic)
Example

```
(nullHeuristic)
```

---

19.4.10 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)
```

---

19.4.11 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)

19.4.12 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)

19.4.13 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)
```

### 19.4.14 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)
```

### 19.4.15 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)
19.5 Heuristics - Transformations

The *transformations* metadata items specify how to normalise the heuristics results into a useful range.

### 19.5.1 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.

---

### 19.5.2 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.
19.5.3 \textbf{logisticFunction}

Transforms heuristic scores by applying the logistic function to them: \( f(x) = \frac{1}{1 + \exp(x)} \).

\textbf{Format}

\begin{verbatim}
(logisticFunction)
\end{verbatim}

\textbf{Example}

\begin{verbatim}
(logisticFunction)
\end{verbatim}

\textbf{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.

\hline

19.5.4 \textbf{tanh}

Transforms heuristic scores by applying the \text{tanh} to them: \( f(x) = \text{tanh}(x) \).

\textbf{Format}

\begin{verbatim}
(tanh)
\end{verbatim}

\textbf{Example}

\begin{verbatim}
(tanh)
\end{verbatim}

\textbf{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.
19.6 Misc

The **misc** package includes miscellaneous items relevant to the AI settings.

19.6.1 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”.

19.6.2 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
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.

20.1 Example

For example, the game description for Breakthrough contains the following define:

\[
(\text{define} \ "\text{ReachedTarget}\" \ (\text{in} \ (\text{lastTo}) \ (\text{region} \ \text{Mover})))
\]

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:

\[
(\text{end} \ (\text{if} \ (\"\text{ReachedTarget}\") \ (\text{result} \ \text{Mover} \ \text{Win})))
\]

This concept is used in many games, all of which can reuse this define to simplify and clarify their own descriptions.
20.2 Parameters

Ludii defines 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 defines 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 defines 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)))
```

20.3 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 defines 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)))
```
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”.

### 20.4 Known Defines

External defines 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 19.
21
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.

21.1 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)

21.2 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.

21.3 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 (hex Diamond <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 (move Add (to (sites Empty))))
   (end (if (is Connected 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.
If the user selects the “Board Size/9x9” and “End Rules > Misère” 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)))
 )
 )
```
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:

\[
\text{(rulesets } \{ \\
\quad \text{ruleset "Ruleset/Name A" } \{ \text{"Option A/Item M" } \text{"Option B/Item N" } \ldots \} \\
\quad \text{ruleset "Ruleset/Name B" } \{ \text{"Option C/Item O" } \text{"Option D/Item P" } \ldots \} \} \ldots \}
\]

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.

### 22.1 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.

\[
\text{(game "Seega" } \\
\quad \text{(players 2) } \\
\quad \text{(equipment } \{ \text{(board (square <Board:size>)) } \ldots \} \})
\]
(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.
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}))

### 23.1 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" \ldots \"D5"\}

will expand to:

\{"B2" \"B3" \"B4" \"B5" \"C2" \"C3" \"C4" \"C5" \"D2" \"D3" \"D4" \"D5"\}
A number of pre-defined constants can be used in game descriptions. These are instantiated with their actual values at compile time.

24.1 Off

Denotes an off-board position.
Internal value: -1

Example

\[(\text{not } (= \text{where } "\text{King}" \text{ Next} \text{ Off}))\]

24.2 End

Denotes the end of a track.
Internal value: -2

Example

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

24.3 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.
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 v1.1.19.

<table>
<thead>
<tr>
<th>Animals</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bear.svg</td>
<td>dog-alt1.svg</td>
</tr>
<tr>
<td>bull.svg</td>
<td>dog.svg</td>
</tr>
<tr>
<td>camel-alt1.svg</td>
<td>dove.svg</td>
</tr>
<tr>
<td>camel-alt2.svg</td>
<td>dragon.svg</td>
</tr>
<tr>
<td>camel-alt3.svg</td>
<td>duck-alt1.svg</td>
</tr>
<tr>
<td>camel.svg</td>
<td>duck.svg</td>
</tr>
<tr>
<td>cat-alt1.svg</td>
<td>eagle.svg</td>
</tr>
<tr>
<td>cat.svg</td>
<td>elephant-alt1.svg</td>
</tr>
<tr>
<td>chick.svg</td>
<td>elephant-alt2.svg</td>
</tr>
<tr>
<td>chicken.svg</td>
<td>elephant.svg</td>
</tr>
<tr>
<td>cow.svg</td>
<td>fish.svg</td>
</tr>
<tr>
<td>coyote-alt1.svg</td>
<td>fox.svg</td>
</tr>
<tr>
<td>coyote.svg</td>
<td>goat-alt1.svg</td>
</tr>
<tr>
<td>crab.svg</td>
<td>goat-alt2.svg</td>
</tr>
<tr>
<td>crocodile.svg</td>
<td>goat-alt3.svg</td>
</tr>
<tr>
<td></td>
<td>goat.svg</td>
</tr>
<tr>
<td></td>
<td>goose.svg</td>
</tr>
</tbody>
</table>
Language Reference

army/Buildings
- airport.svg
- anvil.svg
- barn.svg
- base.svg
- dock.svg
- factory.svg
- farm.svg
- fishing.svg
- generator.svg
- lighthouse.svg

army/Misc
- cannonOutline.svg
- family.svg
- nuclearBomb.svg
- regimentalOutline.svg
- riderOutline.svg
- rocketLaunch.svg
- tinSoldier.svg
- tools.svg
- wagonOutline.svg
- wildWolf.svg

army/P1
- antiair.svg
- artillery.svg
- battleship.svg
- bomber.svg
- boss.svg
- builder.svg
- cruiser.svg
- demolisher.svg
- fighter.svg
- helicopter.svg
- launcher.svg
- motorbike.svg
- shooter.svg
- soldier.svg
- speeder.svg
- submarine.svg
- tank.svg
army/P2
- antiairFlip.svg
- artilleryFlip.svg
- battleshipFlip.svg
- bomberFlip.svg
- bossFlip.svg
- builderFlip.svg
- cruiserFlip.svg
- demolisherFlip.svg
- fighterFlip.svg
- helicopterFlip.svg
- launcherFlip.svg
- motorbikeFlip.svg
- shooterFlip.svg
- soldierFlip.svg
- speederFlip.svg
- submarineFlip.svg
- tankFlip.svg

galley.svg
- robotWarrior.svg
- tinSolider.svg
- zeppelin.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_microsof.svg
- king_microsof.svg
- knight_microsof.svg
- pawn_microsof.svg
- queen_microsof.svg
- rook_microsof.svg

chess/plain
- bishop.svg
- king.svg
- knight.svg
- pawn.svg
- queen.svg
- rook.svg

copyright
- Ludii
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
  symbolaneutral.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

hands
  hand0.svg
  hand1.svg
  hand2.svg
  hand3.svg
  hand4.svg
  hand5.svg
  handflip0.svg
  handflip1.svg
  handflip2.svg
  handflip3.svg
  handflip4.svg
  handflip5.svg
  paper.svg
  rock.svg
  scissors.svg

hieroglyphs
  2flags.svg
  2human.svg
  2human_knee.svg
  2lines.svg
  3ankh.svg
  3ankh_side.svg
  3bird.svg
  3cross.svg
  3flags.svg
  3lines.svg
  3nefer.svg
  ankh_waset.svg
  horus.svg
  senetcross.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

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
- CircleFive.svg
- CircleFour.svg
- CircleNine.svg
- CircleOne.svg
- CircleSeven.svg
- CircleSix.svg
- CircleThree.svg
- CircleTwo.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
arrow.svg
bean.svg
bike.svg
boy.svg
bread.svg
car.svg
castle.svg
castle_alt.svg
cone.svg
corn.svg
doors.svg
dot.svg
eyptLion.svg
fan.svg
fire.svg
flower.svg
flowerHalf1.svg
flowerHalf2.svg
heptagon.svg
human.svg
minotaur.svg
oldMan.svg
paddle.svg
pawn3d.svg
robot.svg
roundPawn.svg
rubble.svg
starFour.svg
stick.svg
sun.svg
teseus.svg
urpiece.svg
waves.svg

ploy
Commander.svg
LanceT.svg
LanceW.svg
LanceY.svg
ProbeBigV.svg
ProbeL.svg
ProbeMinV.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
minus.svg
none.svg
octagon.svg
pentagon.svg
pyramid.svg
rectangle.svg
rectangleVertical.svg
square-alt1.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
narikiyo_study.svg
oshos1_study.svg
oshos1_study.svg
ryuuwa_study.svg
ryuuo_study.svg
tokin_study.svg

**shogi/traditional**

fuhyo.svg
ginsho.svg
hisha.svg
kakugyo.svg
keima.svg
kinsho.svg
kyosha.svg
narigin.svg
narikiyo.svg
narikiyo.svg
oshos1.svg
oshos1.svg
ryuuwa.svg
ryuuo.svg
tokin.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

**toolButtons**

button-about.svg

button-dots-c.svg

button-dots-d.svg

button-dots.svg

button-end-a.svg

button-end.svg

button-match-end.svg

button-match-start.svg

button-next.svg

button-pass.svg

button-pause.svg

button-play.svg

button-settings-a.svg

button-settings-b.svg

button-start-a.svg

button-start.svg

button-swap.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

**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
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 20 for details on the define syntax.

## B.1 def/board

### B.1.1 “AlquerqueBoard”

Defines a rectangular board with Alquerque-style alternating diagonals.

**Example**

```
("AlquerqueBoard" 5 5)
```

```lisp
(define "AlquerqueBoard"
  (board
   (rectangle #1 #2 diagonals:Alternating)
   use:Vertex
  )
)
```

402
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

```scheme
("HoundsAndJackalsBoard")
```

```scheme
(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)
  )

use:Vertex
)
```

B.1.3 “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}}
    )
    use: Vertex
  )
)

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 Cell 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 or The specific moves used to threat. This parameter is optional. If not specify the current location of the piece is used and all the legal moves are used.
Example

```
("IsInCheck" "Osho" Next)
("IsInCheck" "King" Mover at:(to))
```

```lisp
(define "IsInCheck"
  (is Threatened (id #1 #2) #3)
)
```

---

**B.2.3 “NewTurn”**

Checks if this is new turn, i.e. the current move is not the previous mover.

Example

```
("NewTurn")
```

```lisp
(define "NewTurn"
  (not (is Prev 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)
```

```lisp
(define "NoPiece"
  (= (count Pieces #1) 0)
)
```

---

**B.2.5 “NoSites”**

Checks if a region has no sites.

#1 = The region.
Example

("NoSites" (sites Empty))

(define "NoSites"
   (\(= (\text{count} \text{Sites in:#1}) 0)\))

B.2.6 “SameTurn”

Checks if the previous mover is the same player than the current mover, consequently this is the same turn.

Example

("SameTurn")

(define "SameTurn"
   (\text{is Prev Mover})
)

B.3 def/functions

B.3.1 “OccupiedNbors”

Returns the number of pieces owned by a player around a given site.

#1 = The site to test around.
#2 = The owner of the pieces to count.

Example

("OccupiedNbors" (to) Mover)

(define "OccupiedNbors"
   (\text{count} \text{Sites in:(intersection}}
    \text{sites Around #1)}
    \text{sites Occupied by:#2})
)
B.4  def/rules

B.5  def/rules/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 (no Moves Next) (result Next #1))
)

B.6  def/rules/play

B.7  def/rules/play/moves

B.7.1  “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"
  (move Hop
    (between
      if:(is Enemy (who at:(between)))
        (apply (remove (between)))
    )
    (to if:(is Empty (to)))
  )
)
B.7.2 “HopDiagonalCapture”

Defines a hop move in all the diagonal directions over an enemy to an empty site. The enemy piece is removed.

Example

("HopDiagonalCapture")

(define "HopDiagonalCapture"
  (move Hop
    Diagonal
    (between
      if:(is Enemy (who at:(between)))
      (apply (remove (between)))
    )
    (to if:(is Empty (to)))
  )
)

B.7.3 “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 Rows) after:(count Rows) at:EndOfTurn)

(define "HopDiagonalSequenceCapture"
  (move Hop
    Diagonal
    (between
      #1
      #2
      if:(is Enemy (who at:(between)))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
  )
)
(then
  (if (can Move
    (hop
      (from (last To))
      Diagonal
      (between
        #1
        #2
        if:(and
          (not (is In (between) (sites ToClear)))
          (is Enemy (who at:(between)))
        )
      )
      (to if:(is Empty (to)))
    )
  )
  (moveAgain)
)
)
)

### B.7.4 “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 Rows) after:(count Rows) at:EndOfTurn)
```

```lisp
(define "HopDiagonalSequenceCaptureAgain"
  (move Hop
    (from (last To))
    Diagonal
    (between
      #1
      #2
      if:(and (not (is In (between) (sites ToClear))) (is Enemy (who at:(between))))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
  )
)```
B.7.5 “HopOrthogonalCapture”
Defines a hop move in all the orthogonal directions over an enemy to an empty site. The enemy piece is removed.

Example

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

B.7.6 “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**

```lisp
("HopOrthogonalSequenceCapture")
("HopOrthogonalSequenceCapture" before:(count Rows) after:(count Rows) at:EndOfTurn)
```

```lisp
(define "HopOrthogonalSequenceCapture"
  (move Hop
      Orthogonal
      (between #1 #2
        if:(is Enemy (who at:(between)))
        (apply (remove (between) #3))
      )
      (to if:(is Empty (to)))
      (then
        (if (can Move
          (hop
            (from (last To))
            Orthogonal
            (between #1 #2
              if:(and
                (not (is In (between) (sites ToClear)))
                (is Enemy (who at:(between)))
              )
              (to if:(is Empty (to)))
            )
            (moveAgain)
          )
        )
      )
  )
)
```

**B.7.7 “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

```lisp
("HopOrthogonalSequenceCaptureAgain")
("HopOrthogonalSequenceCaptureAgain" before:(count Rows) after:(count Rows) at:EndOfTurn)
```

```lisp
(define "HopOrthogonalSequenceCaptureAgain"
  (move Hop
    (from (last To))
    Orthogonal
    (between
      #1
      #2
      if:(and (not (is In (between) (sites ToClear))) (is Enemy (who at:(between))))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
    (if (can Move
      (hop
        (from (last To))
        Orthogonal
        (between
          #1
          #2
          if:(and
            (not (is In (between) (sites ToClear)))
            (is Enemy (who at:(between)))
          )
        )
        (to if:(is Empty (to)))
      )
      (moveAgain)
    )
  )
)
```

B.7.8 “HopRotationalCapture”

Defines a hop move in all the rotational directions over an enemy to an empty site. The enemy piece is removed.
Example

"HopRotationalCapture"

(define "HopRotationalCapture"
  (move Hop
    Rotational
    (between
      if:(is Enemy (who at:(between)))
      (apply (remove (between)))
    )
    (to if:(is Empty (to)))
  )
)

B.7.9 “HopRotationalSequenceCapture”

Defines a sequence of hop move in all the rotational 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

"HopRotationalSequenceCapture"

(define "HopRotationalSequenceCapture"
  (move Hop
    Rotational
    (between
      #1
      #2
      if:(is Enemy (who at:(between)))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
    (then
      (if (can Move
        (hop
          (from (last To))
          Rotational
          (between
            #1
            #2
            if:(and
              (not (is In (between) (sites ToClear)))
            )
          )
        )
      )
    )
  )
)
B.7.10 "HopRotationalSequenceCaptureAgain"

Defines a sequence of hop move in all the rotational 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

(define "HopRotationalSequenceCaptureAgain"
  (move Hop
    (from (last To))
    Rotational
    (between
      #1
      #2
      if:(and (not (is In (between) (sites ToClear))) (is Enemy (who at:(between))))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
    (then
      (if (can Move
        (hop
          (from (last To))
          Rotational
          (between
            #1
            #2
            if:(and
              (not (is In (between) (sites ToClear)))
              (is Enemy (who at:(between))))
          )
        )
      )
      )
    )
  )
)

(is Enemy (who at:(between)))
)}

(to if:(is Empty (to)))
}

(moveAgain)
)
)}
B.7.11 “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 Rows) after:(count Rows) at:EndOfTurn)

(define "HopSequenceCapture"
  (move Hop
    (between
      #1
      #2
      if:(is Enemy (who at:(between)))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
    (then
      (if (can Move
        (hop
          (from (last To))
          (between
            #1
            #2
            if:(and
              (not (is In (between) (sites ToClear)))
              (is Enemy (who at:(between)))
            )
          )
        )
      )
      (to if:(is Empty (to)))
    )
  )
)
B.7.12 “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 Rows) after:(count Rows) at:EndOfTurn)

(define "HopSequenceCaptureAgain"
  (move Hop
    (from (last To))
    (between
      #1
      #2
      if:(and (not (is In (between) (sites ToClear))) (is Enemy (who at:(between))))
      (apply (remove (between) #3))
    )
    (to if:(is Empty (to)))
    (then
      (if (can Move
          (hop
            (from (last To))
            (between
              #1
              #2
              if:(and
                (not (is In (between) (sites ToClear)))
                (is Enemy (who at:(between)))
              )
            )
            (to if:(is Empty (to)))
          )
        )
      (moveAgain)
    )
  )
)
B.7.13 “StepBackwardToEmpty”
Defines a step move to the backward direction according to the facing direction of the piece in the 'from' location to an empty site.

Example

```
("StepBackwardToEmpty")
```

```
(define "StepBackwardToEmpty"
  (move Step Backward
   (to if:(is Empty (to)))
  )
)
```

B.7.14 “StepDiagonalToEmpty”
Defines a step move to all the diagonal directions to an empty site.

Example

```
("StepDiagonalToEmpty")
```

```
(define "StepDiagonalToEmpty"
  (move Step Diagonal (to if:(is Empty (to)))
  )
)
```

B.7.15 “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"
  (move Step Forwards (to if:(is Empty (to)))
  )
)
```
B.7.16 “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

```lisp
("StepForwardToEmpty")
```

```lisp
(define "StepForwardToEmpty"
  (move Step
    Forward
    (to if:(is Empty (to)))))
)
```

B.7.17 “StepOrthogonalToEmpty”
Defines a step move to all the orthogonal directions to an empty site.

Example

```lisp
("StepOrthogonalToEmpty")
```

```lisp
(define "StepOrthogonalToEmpty"
  (move Step Orthogonal (to if:(is Empty (to)))))
)
```

B.7.18 “StepRotationalToEmpty”
Defines a step move to all the rotational directions to an empty site.

Example

```lisp
("StepRotationalToEmpty")
```

```lisp
(define "StepRotationalToEmpty"
  (move Step Rotational (to if:(is Empty (to)))))
)
```

B.7.19 “StepToEmpty”
Defines a step move to all the adjacent directions to an empty site.

Example
B.8  def/walk

B.8.1  “DominoWalk”
Defines a graphics turtle walk to build a domino as a large piece.

Example

(define "DominoWalk"
  { {F R F R F L F L F R F R F} }
)

B.8.2  “GiraffeWalk”
Defines a graphics turtle walk to get the locations where can move a Chess giraffe.

Example

(define "GiraffeWalk"
  { {F F F R F F} {F F F L F F} }
)

B.8.3  “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.8.4 “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} }
)

B.8.5 “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} }
)
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 t State type, allocating required memory, initialising necessary variables, etc.
C.2 Listing

```plaintext
<game> ::= (game <string> <players> [mode] <equipment> <rules.rules>) |
         (game <string>) | <match>
```

```plaintext
<players> ::= (players <int>) | (players {<players.player>})
<players.player> ::= (player <directionFacing>)
```

```plaintext
<effect> ::= <effect.add> | <addScore> | <attract> | <avoidStoredState> | 
           <bet> | <claim> | <custodial> | <effect.deal> | <directional> | 
           <do> | <enclose> | <firstMoveOnTrack> | <flip> | 
           <operators.forEach> | <forget> | <fromTo> | <hop> | 
           <intervene> | <leap> | <max.max> | <moveAgain> | <note> | 
           <pass> | <playCard> | <priority> | <promote> | <propose> | 
           <push> | <remember> | <effect.remove> | <roll> | <satisfy> | 
           <select> | <effect.set.set> | <shoot> | <slide> | <sow> | 
           <effect.step> | <surround> | <swap.swap> | <take> | <trigger> | 
           <vote> | <while>
```

```plaintext
<apply> ::= (apply if:<boolean>) | (apply <nonDecision>) |
           (apply if:<boolean> <nonDecision>)
```

```plaintext
<attract> ::= (attract [moves.from] [absoluteDirection] [when])
<claim> ::= (claim [moves.piece] [moves.to] [when])
<custodial> ::= (custodial [moves.player] [roleType] [range])
<directional> ::= (directional [moves.from] [direction] [moves.to] [when])
<effect.add> ::= (add [moves.piece] [moves.to] [count:int] [stack:boolean] [when])
<effect.deal> ::= (deal [dealableType] [count:int] [beginWith:int])
<effect.remove> ::= (remove [siteType] [int] [level:int] [at:whenType] [count:int] [when])
<effect.step> ::= (step [moves.from] [direction] [moves.to] [stack:boolean] [when])
<enclose> ::= (enclose [siteType] [moves.from] [direction] [moves.between] [numException:int] [when])
<flip> ::= (flip [siteType] [int] [when])
<fromTo> ::= (fromTo [moves.from] [moves.to] [count:int] [copy:boolean] [stack:boolean] [roleType])
<hop> ::= (hop [moves.from] [direction] [moves.between] [moves.to] [stack:boolean] [when])
<intervene> ::= (intervene [moves.from] [absoluteDirection] [moves.between] [moves.to] [when])
<leap> ::= (leap [moves.from] [{stepType}] [forward:boolean] [rotations:boolean] [moves.to] [when])
<note> ::= (note [player:int] [player:roleType] [string] [int] [ints] [float] [boolean] [sites] [range] [direction] [graph] [to:moves.player] [to:roleType])
<pass> ::= (pass [when])
<playCard> ::= (playCard [when])
<promote> ::= (promote [siteType] [int] [moves.piece] [moves.player] [roleType] [when])
```
<propose> ::= (propose (<string> | {<string>}) [then])
<push> ::= (push [moves.from] <direction> [then])
<random> ::= (random {float} {moves})
<roll> ::= (roll [then])
<satisfy> ::= (satisfy (<boolean> | {<boolean>}))
<select> ::= (select [moves.from] [moves.to] [then])
<shoot> ::= (shoot moves.piece [moves.from] [absoluteDirection] [moves.between] [moves.to] [stack:boolean]
<slide> ::= (slide [moves.from] [string] [direction] [moves.between] [moves.to] [stack:boolean]
<sow> ::= (sow [siteType] [int] [count:int] [string] [owner:int] [if:boolean] [apply:nonDecision]
<surround> ::= (surround [moves.from] [relationType] [moves.between] [moves.to] [except:int] [with:int]
<then> ::= (then nonDecision [applyAfterAllMoves:boolean])
<trigger> ::= (trigger string (int | roleType) [then])
<vote> ::= (vote string | {string}) [then])

// game.rules.play.moves
<moves> ::= <effect.add | <addScore | <allCombinations | <logical.and |<append> | <apply> | <attract> | <avoidStoredState> | <bet> |<claim> | <custodial> | <effect.deal> | <decision> |<directional> | <do> | <effect> | <enclose> |
<firstMoveOnTrack> | <flip> | <operators.foreach.forEach> |
<forget> | <fromTo> | <hop> | <logical.if> | <intervene> |
<leap> | <max.max> | <move> | <moveAgain> | <decision> |
<note> | <operator> | <logical.or> | <pass> | <playCard> |
<priority> | <promote> | <propose> | <push> | <random> |
<remember> | <effect.remove> | <roll> | <satisfy> | <select> |
<effect.set.set> | <shoot> | <slide> | <sow> | <effect.step> |
<surround> | <swap.swap> | <take> | <trigger> | <vote> | <while>

// game.rules.play.moves.decision
<decision> ::= <move>
<move> ::= (move Slide [moves.from] [string] [direction] [moves.between] [moves.to] [stack:boolean]
| (move Shoot <moves.piece> [moves.from] [absoluteDirection] [moves.between] [moves.to] [then]) |
| (move Select <moves.from> [moves.to] [then]) |
| (move Promote [siteType] [int] <moves.piece> [moves.player] [roleType] [then]) |
| (move Swap Pieces [int] [int] [then]) |
| (move Remove [siteType] [int] |
| (sites) [level:int] [at:whenType] [count:int] [then]) |
| (move Set TrumpSuit <int> |
| (intArray.math.difference) [then]) |
| (move Set NextPlayer (moves.player) [ints] [then]) |
| (move Set Rotation [moves.to] [int] |
| [previous:boolean] [next:boolean] [then]) |
| (move Step [moves.from] [direction] <moves.to> [stack:boolean] [then]) |
| (move Swap Players (int | roleType) (int | roleType) [then])
(move <moveSiteType> [<moves.piece>] <moves.to> [count:<int>] [stack:<boolean>] [then]])
| (move Bet (<moves.player> | <roleType>) <range> [then]) |
| (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]) |
| (move <moves.from> <moves.to> [count:<int>] [copy:<boolean>] [stack:<boolean>] [roleType] [then])

<moveMessageType> ::= Propose | Vote
<moveSimpleType> ::= Pass | PlayCard
<moveSiteType> ::= Add | Claim

// 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])
<while> ::= (while <boolean> <moves> [then])

// game.rules.play.moves.nonDecision.effect.requirement.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 <setPlayerType> (<moves.player> | <roleType>) <int> [then]) |
| (set Pending [int] | <sites> [then]) |
| (set Var [string] [int] [then]) |
| (set Hidden [hiddenData] | {<hiddenData>} | [siteType] [at:<int> | <sites> [level:<int>] [boolean] [to:<moves.player> | to:<roleType> [then]] | (set TrumpSuit [int] | [intArray.math.difference] [then]) | (set NextPlayer [<moves.player> | <ints> [then]] | (set Rotation [<moves.to>] [[<int> | <int>] [previous:<boolean>] [next:<boolean> [then]]) | (set Team <int> [roleType] [then]) |
| (set <setValueType> [int] [then]) |
| (set <setSiteType> [siteType] at:<int> [level:<int>] [int] [then])
<setPlayerType> ::= Score | Value
<setSiteType> ::= Count | State | Value
<setValueType> ::= Counter | Pot

// game.rules.play.moves.nonDecision.effect.state

<addScore> ::= (addScore {{<int>} | {<roleType>}} {{<int>} [then]}) |
(addScore <moves.player> | <roleType> | <int> [then])
<moveAgain> ::= (moveAgain [then])

// game.rules.play.moves.nonDecision.effect.state.forget

<forget> ::= (forget Value [string] int [then]) |
(forget Value [string] All [then])

// game.rules.play.moves.nonDecision.effect.state.remember

<remember> ::= (remember State [then]) |
(remember Value [string] int [unique:boolean] [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>) [at:<int> | to:<sites> | <siteType>] [then]) |
(take Domino [then])

// game.rules.play.moves.nonDecision.operators.foreach

<operators.foreach.forEach> ::= (forEach Site <sites> <moves> [noMoveYet:<moves>] [then]) |
(forEach Value <ints> <moves> [then]) |
(forEach Value min:<int> max:<int> <moves> [then]) |
(forEach Piece [on:<siteType>] [string] | 
{string}] [container:<int> | 
<moves> [<moves.player>] | 
<roleType>] [top:<boolean>] [then]) |
(forEach Team <moves> [then]) |
(forEach Group [siteType] [direction] [if:boolean] <moves> [then]) |
(forEach Die [int] [combined:boolean] [replayDouble:boolean] [if:boolean] <moves> [then]) |
(forEach Direction [<moves.from>] [direction] [<moves.between>] [moves.to])
```plaintext
| <moves> [then] | (forEach Player <moves> [then]) | (forEach <ints> <moves> [then]) | (forEach Level [siteType] <int> <moves> [then])

// 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}) [equipment.values] | {equipment.values} [use:siteType] | boardless | mancalaBoard | surakartaBoard
board.track ::= (track [string] {{int}} | string) | {equipment.trackStep} [loop:boolean] [int] | roleType [directed:boolean])
boardless ::= (boardless [tilingBoardlessType] [dim])

// game.equipment.container.board.custom

mancalaBoard ::= (mancalaBoard <int> <int> [store:storeType] [numStores:int] [board.track] | {board.track})
```

<surakartaBoard> ::= (surakartaBoard <graph> [loops:<int>] [from:<int>])

// game.mode
<mode> ::= (mode <modeType>)

// game.equipment
<equipment> ::= (equipment {<item>})
 <item> ::= <component> | <container> | <dominoes> | <hints> | <map> | <regions>

// game.equipment.component
<component> ::= (component <string> <roleType> {{<stepType>}} <directionFacing> <moves> <int> <int> <int>)
 <component.card> | <die> | <domino> | <component.piece> |
 <tile>
<component.card> ::= (card <string> <roleType> <cardType> rank:<int> value:<int> trumpRank:<int> trumpValue:<int> suit:<int> [moves] [maxState:<int>] [maxCount:<int>] [maxValue:<int>])
<component.piece> ::= (piece <string> [roleType] [directionFacing] [flips] [moves] [maxState:<int>] [maxCount:<int>] [maxValue:<int>])
<die> ::= (die <string> <roleType> numFaces:<int> [directionFacing] [moves])

// game.equipment.component.tile
<tile> ::= (tile <string> [roleType] [{<stepType>} | slots:{<int>} | numSides:<int> | slotsPerSide:<int> | [numPaths:<int>] [flips] [moves] [maxState:<int>] [maxCount:<int>] [maxValue:<int>])
<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> | <other.hand>

// game.equipment.container.other
<deck> ::= (deck [roleType] [cardsBySuit:<int>] [suits:<int>] [{equipment.card}])
<dice> ::= (dice [d:<int>] [faces:{<int>}] [facesByDie:{<int>}] | from:<int>] [roleType] num:<int> [biased:{<int>}]])
<other.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)
<gravity> ::= (gravity [PyramidalDrop])
<meta.swap> ::= (swap)
<metaRule> ::= <automove> | <gravity> | <noRepeat> | <passEnd> | <pin> | <meta.swap>
<noRepeat> ::= (noRepeat [repetitionType])
<pin> ::= (pin SupportMultiple)

// game.rules.start

<start> ::= (start ({<startRule>} | <startRule>))
<start.deal> ::= (deal dealableType [int])
<startRule> ::= <start.deal> | <forEach.forEach> | <place> | <deductionPuzzle.set> | <start.set.set> | <split>

// game.rules.start.deductionPuzzle

<deductionPuzzle.set> ::= (set [siteType] {<int>})

// game.rules.start.forEach

<forEach.forEach> ::= (forEach ints <startRule>) | (forEach Player <startRule>) | (forEach Value min:<int> max:<int> <startRule>) | (forEach Site sites [if:<boolean>] <startRule>) | (forEach Team <startRule>)

// game.rules.start.place

(place Random {count} int [siteType]) | (place Random string) [count] [state:int] [value:int] int [siteType])
| (place Random <sites> string) [count] [state:int] [value:int] [siteType])
| (place Stack string | sites string) [count] [state:int] [value:int] | (place <string> [siteType] {{<int>}} sites {{<string>}} counts:{<int>} [state:int] |
| (place <string> [siteType] {{<int>}} sites {{<string>}} counts:{<int>} [state:int] |

// game.rules.start.set
<start.set.set> ::= (set <setStartPlayerType> [roleType] <int>) | (set <setStartSitesType> <int> [siteType] (at:<int> | to:<sites>)) | (set <roleType> [siteType] [int] [coord:<string>]) | (set <roleType> [siteType] [int] [level:<int>] [boolean] to:<roleType>) | (set Team <int> {roleType}) | (set RememberValue [string] (int | <sites>) [level:<int>] [boolean] to:<roleType>) | (set Hidden [hiddenData] | {hiddenData} [siteType] (at:<int> | <sites>) [level:<int>] [boolean] to:<roleType>) |

<setStartPlayerType> ::= Amount | Score
<setStartSitesType> ::= Cost | Count | Phase

// game.rules.start.split

<split> ::= (split Deck)

// game.rules.play

<play> ::= (play <moves>)

// game.rules.end

<byScore> ::= (byScore {<end.score>})
<end.forEach> ::= (forEach [roleType] if:boolean <result>)
<end.if> ::= (if boolean [end.if] [<result>])
<endRule> ::= <end.forEach> | <end.if>
<payoffs> ::= (payoffs {payoff})
<result> ::= (result roleType <resultType>) | <byScore> | <payoffs>

// game.match

<match> ::= (match string [players] <games> end) | (match string)
<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> (<int> | <roleType>))
<< ::= (< <int> <int>)
<> ::= (= <sites> <sites>) | (= <int> (<int> | <roleType>))
=> ::= (> <int> <int>)
>= ::= (>= <int> <int>)

<booleans.math.if> ::= (if <boolean> <boolean> [if: <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 <isConnectType> [int] [siteType] [at: <int>] [direction] [sites] | roleType | regionTypeStatic)) |
   (is Line [siteType] <int> [absoluteDirection] [through: <int> | throughAny: <sites>] [roleType] | what: <int> |
    what: <int>]) [exact: <boolean> | contiguous: boolean] [if: boolean] [byLevel: <boolean>])
   (is Target [int] | string) {<int>} [<int>] [at: <int>] [region] |
   (is Related <relationType> [siteType] <int> (int | sites)) |
   (is <isComponentType> [int] [siteType] [at: <int>] | in: <sites> | moves]) |
   (is Hidden [hiddenData] [siteType] at: <int> [level: <int>] (to: <moves.player> |
   to: <roleType>)) | (is In [int] | int) [siteType] | <int> |<sites> |
   (is Path <siteType> [from: <int>] <moves.player> |
    roleType]) [length: range] [closed: boolean]) |
   (is Loop [siteType] surround: roleType) |
   {roleType} [direction] [int] [int] |
   (is <isIntegerType> [int]) |
   (is componentType) |
   (is Triggered <string> (<int> | roleType))
<isComponentType> ::= Threatened | Within
<isConnectType> ::= Blocked | Connected
<isGraphType> ::= LastFrom | LastTo
<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.booleans.can
<can> ::= (can Move <moves>)

// game.functions.region.sites
<sites> ::= (sites [siteType] {string}) |
| (sites <sitesSimpleType> [siteType]) |
| (sites <sitesEdgeType>) |
| (sites <sitesIndexType> [siteType] [int]) |
| (sites <siteType> [int] {{<stepType>}} [rotations:<boolean>]) |
| (sites {{int} | <ints>}) | (sites <sitesMoveType> <moves>) |
| (sites Between [direction] [siteType] from:<int> [fromIncluded:<boolean>] to:<int> [toIncluded:<boolean>] [cond:<boolean>]) |
| (sites) |
| (sites Hidden [hiddenData] [siteType] (to:<moves.player> | to:<roleType>)) |
| (sites Pattern {{<stepType>}} [siteType] [from:<int>] [what:<int> | whats:{<int>}] | (sites Group [siteType] (at:<int> | from:<sites>) [direction] [if:<boolean>]) |
| (sites Crossing at:<int> [moves.player] | <roleType>) |
| (sites Random [sites] [num:<int>]) |
| (sites LargePiece [siteType] at:<int>) |
| (sites Occupied (by:<moves.player> | by:<roleType>) [container:<int> | container:<string>] [component:<int> | component:<string> | components:{<string>}] [top:<boolean>] [on:<siteType>]) |
| (sites Incident <siteType> of:<siteType> at:<int> [owner:<moves.player> | <roleType>]) |
| (sites Around [siteType] (<int> |
<sites>) [regionTypeDynamic>] [distance:<int>] [absoluteDirection>] [if:<boolean>] [includeSelf:<boolean>]
| (sites Direction [from:<int>] | from:<sites>) [direction>] [included:<boolean>] [stop:<boolean>] [stopIncluded:<boolean>] [direction]
| (sites LineOfSight [lineOfSightType] [siteType] [at:<int>] [direction]
| (sites Loop [inside:<boolean>] [siteType] [surround:<roleType>] [roleType] [direction] [int] | <sites>] |
| (sites Side [siteType] [moves.player] | roleType] | compassDirection]) |
| (sites Distance [siteType] [relationType] [effect.step] from:<int> range) |
| (sites [moves.player] | roleType] [siteType] [string]) |
| (sites Start <moves.piece>) | (sites Track [moves.player] | roleType] [string] [from:<int>] [to:<int>]) |
| (sites sitesPlayerType [siteType] [moves.player] | roleType] [nonDecision] [string]) |
| (region string <container.board.board> {string}) |
| (region) | (region {int}) | (region) |
| (region.math.difference) | (expand) <region.foreach.forEach> |
| (region.math.if) <region.math.intersection> |
| (region.last.last) <region.math.union> |

<lineOfSightType> ::= Empty | Farthest | Piece
<sitesEdgeType> ::= Angled | Axial | Horizontal | Slash | Slosh | Vertical
<sitesIndexType> ::= Cell | Column | Edge | Empty | Layer | Phase | Row | State
<sitesMoveType> ::= Between | From | To
<sitesPlayerType> ::= Hand | Winning
<sitesSimpleType> ::= Board | Bottom | Centre | ConcaveCorners | ConvexCorners | Corners | Hint | Inner | LastFrom | LastTo | Left | LineOfPlay | Major | Minor | Outer | Pending | Perimeter |Playable | Right | Top

//-----------------------------------------------------------------------------------------------
// game.functions.region.math

<expand> ::= (expand [int] | string] [sites] | origin:<int>] [steps:<int>] [absoluteDirection] [siteType])
<region.math.difference> ::= (difference <sites> <sites> | <int>)
<region.math.if> ::= (if <boolean> <sites> [sites])
<region.math.intersection> ::= (intersection {sites}) | (intersection <sites> <sites>)
<region.math.union> ::= (union {sites}) | (union <sites> <sites>)

//------------------------------------------------------------------------------------------------
// game.functions.region.last

<region.last.last> ::= (last Between)

//------------------------------------------------------------------------------------------------
// game.functions.region.foreach
<region.forEach.forEach> ::= (forEach Team <sites>) | (forEach <sites> if:<boolean>) | (forEach <ints> <sites>)

// game.functions.intArray.values
<values.values> ::= (values Remembered [<string>])

// game.functions.intArray.sizes
<sizes> ::= (sizes Group [<siteType>] [<direction>] [<roleType> | of:<int> | if:<boolean>] [min:<int>])

// game.functions.intArray.players
<intArray.players.players> ::= (players <playersTeamType> [if:<boolean>]) | (players <playersManyType> [of:<int>] [if:<boolean>])
<playersManyType> ::= All | Ally | Enemy | Friend | NonMover
<playersTeamType> ::= Team1 | Team10 | Team11 | Team12 | Team13 | Team14 | Team15 | Team16 | Team2 | Team3 | Team4 | Team5 | Team6 | Team7 | Team8 | Team9

// game.functions.intArray.iterator
<team> ::= (team)

// game.functions.intArray.array
<array> ::= (array <sites>)

// game.functions.ints.value
<value> ::= (value) | (value Piece [<siteType>] at:<int> [level:<int>]) | (value Player (<int> | <roleType>)) | (value Pending) | (value Random <range>)

// 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 [prevType])
<rotation> ::= (rotation [siteType] at:int [level:int])
<state.score> ::= (score <moves.player> | <roleType>)
<var> ::= (var [string])
<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 [Territory] [siteType] <roleType> | moves.player) [absoluteDirection]] | (size [Group] [siteType] at:int [direction] [if:boolean]) | (size LargePiece [siteType] (in:sites | at:int)) | (size Stack [siteType] [in:sites | at:int]) | (size Array <ints>)

// game.functions.ints.math
<%> ::= (% <int> <int>)
<ints.math.*> ::= (* ({<int>} | <ints>)) | (* <int> <int>)
<ints.math.+> ::= (+ ({<int>} | <ints>)) | (+ <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>) | (max <int> <int>)
<ints.math.min> ::= (min <int>) | (min <int> <int>)

// game.functions.ints.match
<matchScore> ::= (matchScore <roleType>)

// game.functions.ints.match
// game.functions.ints.last
<ints.last.last> ::= (last <lastType> [afterConsequence:<boolean>])
<lastType> ::= From | To

// game.functions.ints.iterator
<iterator.between> ::= (between)
<iterator.edge> ::= (edge) | (edge <int> <int>)
<iterator.from> ::= (from [at:<whenType>])
<iterator.hint> ::= (hint [at:<int>])
<iterator.player> ::= (player)
<iterator.to> ::= (to)
<iterator.track> ::= (track)
<level> ::= (level)
<pips> ::= (pips)
<site> ::= (site)

// game.functions.ints.dice
<dice.face> ::= (face <int>)

// game.functions.intArray.math
<intArray.math.difference> ::= (difference <ints> (<ints> | <int>))
<intArray.math.intersection> ::= (intersection {<ints>}) | (intersection <ints> <ints>)
<intArray.math.union> ::= (union {<ints>}) | (union <ints> <ints>)
<results> ::= (results (from:<int> | from:<sites>) (to:<int> | to:<sites>) <int>)

// 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 [vertices:{<float>}] [edges:{<dim>}] [edgesCurved:{<float>}] [cells:{<float>}] [cells:{<dim>}] [connect:<boolean>])
<operators.remove> ::= (remove <graph> <poly> [trimEdges:<boolean>]) | (remove <graph> [cells:{<float>}] |
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.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> | <concentric> | <hex> | <quadhex> | <rectangle> | <spiral> | <square> | <tiling> | <tri> | <wedge>

// game.functions.floats.math
<cos> ::= (cos <float>)
<exp> ::= (exp <float>)
<floats.math.*> ::= (* {<float>}) | (* <float> <float>)
<floats.math.+> ::= (+ {<float>}) | (+ <float> <float>)
<floats.math.-> ::= (- <float> <float>)
<floats.math./> ::= (/ <float> <float>)
<floats.math.^> ::= (^ <float> <float>)
<floats.math.abs> ::= (abs <float>)
<floats.math.max> ::= (max {<float>}) | (max <float> <float>)
<floats.math.min> ::= (min {<float>}) | (min <float> <float>)
<log> ::= (log <float>)
<log10> ::= (log10 <float>)
<sin> ::= (sin <float>)
<sqrt> ::= (sqrt <float>)
<tan> ::= (tan <float>)
// game.functions.floats

<float> ::= <floats.math.*> | <floats.math.+> | <floats.math.-> |
          <floats.math./> | <floats.math.^> | <floats.math.abs> | <cos> |
          <exp> | float | <log> | <log10> | <floats.math.max> |
          <floats.math.min> | <sin> | <sqrt> | <tan> | <toFloat>

<toFloat> ::= (toFloat (<boolean> | <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]>]) | <exact> | <range.math.max> |
          <range.math.min>

// game.functions.intArray.state

<rotations> ::= (rotations (<absoluteDirection> | {<absoluteDirection>}))

// game.functions.ints.count

<count.count> ::= (count Groups [<siteType>] [<direction>] [<roleType> | 
                       of:<int> | if:<boolean> [min:<int>]] | 
                  (count Liberties [<siteType>] [at:<int>] [<direction>] [if:<boolean>] |
                  (count Steps [<siteType>] [<relationType>] [<effect.step>] [<int> |
                       [<sites>]]) | 
                  (count <countComponentType> [<siteType>] [<roleType> | 
                       of:<int>] [string] [in:<sites>]) | 
                  (count [countSiteType] [string] [in:<sites>] [at:<int> | 
                       [string]]) | (count <countSimpleType> [siteType]) | 
                  (count Stack [stackDirection] [siteType] [at:<int> | 
                       to:<sites>] [if:<boolean> [stop:<boolean>]]) | 
                  (count StepsOnTrack [roleType] [moves.player] | 
                       [string] [<int>] [string] [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.card
\[\text{<card.card>} ::= \text{<card <cardSiteType> at:<int> [level:<int>] | <card TrumpSuit>)}
\]
\[\text{<cardSiteType>} ::= \text{Rank | Suit | TrumpRank | TrumpValue)}
\]

// game.functions.ints.board.where

\[\text{<where>} ::= \text{(where Level <int> [siteType]} at:<int> [fromTop:<boolean>]) | \text{(where Level <string> (int} | \text{roleType)} [state:<int>] [siteType]} at:<int> [fromTop:<boolean>])
\]

// game.functions.ints.board

\[\text{<ahead>} ::= \text{(ahead [siteType] <int> [steps:<int>] [direction])}
\]

// game.functions.dim.math

\[\text{<dim.math.*> ::= (* {dim}) | (* <dim> <dim>)}
\]

// game.functions.dim

\[\text{<dim>} ::= \text{<dim.math.*> | <dim.math.+> | <dim.math.-> | <dim.math./> | <dim.math.> | <dim.math.abs> | int | <dim.math.max> | <dim.math.min>)
\]

// game.functions.ints

\[\text{<ints>} ::= \{<int>} | \text{array} | \text{intArray.math.difference)} |
<intArray.math.intersection> | <intArray.players.players> | <regions> | <results> | <rotations> | <sizes> | <team> | <intArray.math.union> | <values.values>

<int> ::= (%< > | <dim.math.*> | <ints.math.*> | <dim.math.+> | <ints.math.+> | <dim.math.-> | <ints.math.-> | <dim.math./> | <ints.math./> | <dim.math.abs> | <dim.math.abs>| <ahead> | <amount> | <iterator.between> | <card.card> | <centrePoint> | <column> | <coord> | <cost> | <count.count> | <counter> | <iterator.edge> | <dice.face> | <iterator.from> | <handSite> | <iterator.hint> | <id> | <ints.math.if> | int | <ints.last.last> | <layer> | <level> | <mapEntry> | <matchScore> | <dim.math.max> | <ints.math.max> | <dim.math.min> | <ints.math.min> | <mover> | <next> | <nextPhase> | <pathExtent> | <board.phase> | <pips> | <iterator.player> | <pot> | <prev> | <regionSite> | <rotation> | <rou> | <state.score> | <site> | <size> | <state> | <iterator.to> | <toInt> | <topLevel> | <iterator.track> | <trackSite> | <value> | <var> | <what> | <where> | <who>

<toInt> ::= (toInt (<boolean> | <float>))

// game.functions.booleans.all
<booleans.all.all> ::= (all <allSitesType> | sites if:booleans) | (all <allSimpleType> | (all Values <ints> if:booleans)

<allSimpleType> ::= DiceEqual | DiceUsed | Passed
<allSitesType> ::= Different | Sites

// game.functions.booleans
<booleans.all.all> ::= (all <allSitesType> <sites if:booleans>)

<booleans.all.all> ::= (all <allSimpleType> | (all Values <ints> if:booleans)

<booleans.is.is> | <booleans.no.no> | <not> | <math.or> | <toInt> | <var> | <xor>

<toInt> ::= (toInt (<int> | <float>))

// game.functions.directions
<directions> ::= (directions (absoluteDirection> | {absoluteDirection>}) | directions [relativeDirection> | {relativeDirection> [of:relationType> | bySite:booleans]) | directions <siteType> from:<int> to:<int>) | (directions Random <direction> num:<int>)

<directions.difference> ::= (difference <direction> <direction>)
<directions.if> ::= (if <boolean> <direction> <direction>)

// game.util.end
<end.score> ::= (score <roleType> <int>)
<payoff> ::= (payoff <roleType> <float>)

// 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) | (graph vertices:{<float>}) [edges:{<int>}] | (graph) | (graph <graph>) | <operators.add> | <basis> | <brick> | <celtic> | <clip> | <complete> | <concentric> | <dual> | <hex> | <hole> | <intersect> | <keep> | <layers> | <makeFaces> | <merge> | <quadhex> | <recoordinate> | <regular> | <operators.remove> | <renumber> | <repeat> | <rotate> | <scale> | <shift> | <skew> | <spiral> | <splitCrossings> | <square> | <subdivide> | <tiling> | <tri> | <trim> | <operators.union> | <wedge>
<poly> ::= (poly {{<dim>}}) | (poly {{<float>}})

// 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> | {<string> | {<int>}}] [state:<int>])
<moves.player> ::= (player <int>)
<moves.to> ::= (to [siteType] [sites] [int] [level:<int>] [rotations] [if:<boolean>] [apply])

// 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.trackStep> ::= (trackStep (<int> [compassDirection] [trackStepType]))
<equipment.values> ::= (values <siteType> <range>)

// game.util.directions
<direction> ::= <absoluteDirection> | directions.difference | <directions>
<directions.if> | <relativeDirection>

<absoluteDirection> ::= Adjacent | All | Angled | Axial | Base | CCW | CW | D |

DE | DN | DNE | DNW | DS | DSE | DW | Diagonal |

Downward | E | ENE | ESE | In | N | NE | NNE | NW | NNW |

OffDiagonal | Orthogonal | Out | Rotational | S | SE | SSE |

SSW | SW | SameLayer | Support | U | UE | UN | UNE | UNW | US |

USE | USW | UW | Upward | W | WW | WSW

<compassDirection> ::= E | ENE | ESE | N | NE | NNE | NNW | NW | S | SE | SSE |

SSW | SW | W | WW | WSW

<directionFacing> ::= <absoluteDirection> | <directions> | <directions.if> | <relativeDirection>

<relativeDirection> ::= BL | BLL | BLLL | BR | BRR | BRRR | Backward |

Backwards | FL | FLL | FLLL | FR | FRR | Forward |

Forwards | Leftward | Leftwards | OppositeDirection |

Rightward | Rightwards | SameDirection

<stackDirection> ::= FromBottom | FromTop

// 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

<passEndType> ::= Draw | NoEnd

<prevType> ::= Mover | MoverLastTurn

<repetitionType> ::= Positional | PositionalInTurn | Situational |

SituationalInTurn

<resultType> ::= Abandon | Crash | Draw | Loss | Tie | Win

<roleType> ::= All | Ally | Each | Enemy | Friend | Mover | Neutral | Next |

NonMover | P1 | P10 | P11 | P12 | P13 | P14 | P15 | P16 | P2 |

P3 | P4 | P5 | P6 | P7 | P8 | P9 | 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 | Concentric | Dual | Hexagonal |

HexagonalPyramidal | Mesh | Morris | NoBasis | QuadHex |

Spiral | Square | SquarePyramidal | T31212 | T333333_33434 |

T33336 | T3344 | T33434 | T3464 | T3636 | T4612 | T488 |
Triangular

<hiddenData> ::= Count | Rotation | State | Value | What | Who
<landmarkType> ::= BottomSite | CentreSite | FirstSite | LastSite | LeftSite | RightSite | Topsite
<puzzleElementType> ::= Cell | Edge | Hint | Vertex
<regionTypeDynamic> ::= 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 | OffDiagonal | Orthogonal
<shapeType> ::= Circle | Cross | Custom | Diamond | Hexagon | Limping | NoShape | Polygon | Prism | Quadrilateral | Rectangle | Regular | Rhombus | Spiral | Square | Star | Triangle | Wedge | Wheel
<siteType> ::= Cell | Edge | Vertex
<stepType> ::= F | L | R
<storeType> ::= Inner | None | Outer
<tilingBoardlessType> ::= Hexagonal | Square | Triangular
<trackStepType> ::= End | Off | Repeat

// game.types

<string> ::= string