Ž

ZRF Language Reference

Introduction

What's New in Version 2

Writing Your Own Zillions Game

Advanced

Advanced Topic FAQ

The Internals of Movement

Keywords

set-attribute	set-flag	set-position-flag
solution	stalemated	start-rectangle
strategy	symmetry	title
to	total-piece-count	translate
true	turn-order	unlink
variant	verify	while
win-condition	win-sound	zone
define	include	version
\	`	;

By Jeff Mallett, Last revision: May 2, 2003

What's New in Version 2

1. New Commands

a) There is a new create keyword for adding pieces to the board.

b) There is a new option keyword, so future game options can be backward-compatible (unknown options can be ignored) and don't clutter the "name space". Many of the available options ("progressive levels", "discard cascades", "smart moves", "silent ? player", "show moves list", "selection screen", "highlight goals") are new to Zillions. Some existing options have been expanded. See the option documentation for details.

c) There is a new goal-position? keyword that allows you to check whether a position is in an absolute-config goal. This is useful in building move code dependent on the goals, but which can be used in multiple variants with differently placed goals. Also, not-goal-position?

2. Deprecated Commands

The following keywords have been subsumed by option and are obviated:

- allow-flipping
- animate-captures
- animate-drops
- include-off-pieces
- maximal captures
- pass-partial
- pass-turn
- recycle-captures
- recycle-promotions

They are still supported for backward-compatibility. Note that allow-flipping is replaced by its opposite, "prevent flipping", so that options will consistently default to off.

Some replacement options have extended capabilities: "prevent flipping" has a setting allowing Flip Board to swap the positions without inverting the board bitmap. "animate captures" and "animate drops" allow you to force animation to be on.

3. Expanded Commands

a) The total-piece-count goal now accepts an optional <piece-type> argument to specify that only certain kind of pieces should be counted.

b) The total-piece-count goal is no longer a "top-level goal". This means it can now be nested within **or**'s, **not**'s, and **and**'s. Also, it can be used in the top-level of a win/loss/draw-condition with a

specified player list.

c) Zillions now allows the use of the opposite keyword with "capture", "change-owner", "change-type", or "flip". For example: (flip (opposite e))

d) open can now take URLs as arguments, e.g. clicking on a piece can open a webpage. http, ftp, and mailto links supported.

4. Extended Limits

Some internal limits were extended or removed altogether. For example, a single move or drop may initiate any number of captures, flips, change-owners, change-types, and set-attributes.

Writing Your Own Zillions Game

The rules for a game are stored in a Zillions Rules File. These files end with a ".zrf". Zillions loads the ZRF file, and uses it to find out how to run the game. ZRF files have 4 main parts: the board, the pieces, the goals of the game, and additional information like help and strategy.

You can use any common text editor (like Windows Notepad) to create your new game. Just remember to save your game with a ".zrf" ending, and make sure you save as text or ASCII.

Framework

Here is a framework for a game:

```
(game
;...players, help and extra information goes here
;...board stuff goes here
;...piece stuff goes here
;...goals of game go here
)
```

There are some important things to note about the overall format. Firstly, you will want to pay careful attention to match **parentheses** when writing Zillions Rules Files. The parentheses are used to group sections together, so use a ")" for every "(" you type.

Notice that some lines begin with a semi-colon (";") character. This indicates that the rest of this line is a **comment** and will be ignored when Zillions loads this ZRF. It is a good idea to use lots of comments in your ZRF to make it easier to understand.

Consistent **indentation** also makes your program easier to read. Zillions, on the other hand, doesnt care. It treats all "white space" equally, from a single space to many tabs and carriage returns. The exception is in a string, delimited with quotation marks, which well get to shortly.

Help Descriptions

OK, let's add a little more to our game:

```
(game
  (title "Tic-Tac-Toe")
  (description "One side takes X's and the other side takes O's.
   Players alternate placing their marks on open spots.
   The object is to get three of your marks in a row horizontally,
   vertically, or diagonally. If neither side accomplishes this,
   it's a cat's game (a draw).")
  (history "Tic-Tac-Toe was an old adaptation of Three Men's Morris    to
   situations where there were no available pieces. You can draw or
   carve marks and they are never moved. It is played all over the
   world under various names, such as 'Noughts and Crosses' in
   England.")
  (strategy "With perfect play, Tic-Tac-Toe is a draw. Against less
```

```
   than perfect opponents it's an advantage to go first, as having an
   extra mark on the board never hurts your position. The center is
   the key square as 4 possible wins go through it. The corners are
   next best as 3 wins go through each of them. The remaining
   squares are least valuable, as only 2 wins go through them.
   Try to get in positions where you can `trap` your opponent by
   threatening two 3-in-a-rows simultaneously with a single move. To
   be a good player, you must not only know how to draw as the second
   player, you must also be able to takes advantage of bad play.")
)
```

You can see we have added several sections which describe the game. The "title" section gives our game a name. All ZRF games need names. A single ZRF can contain multiple games and variants, so dont forget to add a title. In this case, we are programming the game "Tic-Tac-Toe".

Next, you will see sections on "description", "history" and "strategy". This is extra information for the user, to tell them the object of the game, the history of the game, and suggested strategy to improve his play.

The Complete Game

OK, now we need to add some more parts:

```
(game
  (title "Tic-Tac-Toe")
; description, history and strategy omitted to save space.
  (players X O)
  (turn-order X O)
  (board
    (image "images\TicTacToe\TTTbrd.bmp")
    (grid
      (start-rectangle 16 16 112 112) ; top-left position
      (dimensions ;3x3
        ("top-/middle-/bottom-" (0 112)) ; rows
        ("left/middle/right" (112 0))) ; columns
      (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
    )
  )
  (piece
    (name man)
    (help "Man: drops on any empty square")
    (image X "images\TicTacToe\TTTX.bmp"
     O "images\TicTacToe\TTTO.bmp")
    (drops ((verify empty?) add))
  )
  (board-setup
    (X (man off 5))
    (O (man off 5))
  )

  (draw-condition (X O) stalemated)
  (win-condition (X O)
    (or (relative-config man n man n man)
        (relative-config man e man e man)
```

```
        (relative-config man ne man ne man)
        (relative-config man nw man nw man)
    )
  )
)
```

The above sample is a complete game. You could cut it out and paste it into a ZRF file, and play it. Let's look at each of the new sections:

Players

```
(players X O)
```

This line tells Zillions the names of the players. In this case, there are two players named "X" and "O". That was easy!

```
(turn-order X O)
```

This line tells Zillions the turn-order that players move. In this case, "X" will move first, and then "O" will move. After that, the turn order will repeat, so "X" will move again, and so on.

OK, we have told Zillions about the players. Lets tell the program about the board:

Board

```
  (board
    (image "images\TicTacToe\TTTbrd.bmp")
    (grid
      (start-rectangle 16 16 112 112) ; top-left position
      (dimensions ;3x3
        ("top-/middle-/bottom-" (0 112)) ; rows
        ("left/middle/right" (112 0))) ; columns
      (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
    )
  )
```

Use the "board" statement to tell Zillions about the board. The "image" statement tells Zillions what bitmap file (*.BMP) to use to display the board. In this case, the file in "images\TicTacToe\TTTbrd.bmp" will be used. You can create new board images using a graphics editor, like Windows Paint.

The "grid" statement makes it easy for you to specify a regularly spaced set of positions. The "start-rectangle" tells Zillions rectangle of the upper left screen position. In this case, this is from the point 16,16 to the point 112, 112. Points are measured over and down from the upper left corner of the window.

The "dimensions" section has information about the placement and name of the positions. The line " ("top-/middle-/bottom-" (0 112)) " tells Zillions the rows will be named "top-", "middle-" and "bottom-". To progress downward to the next row, the Y coordinate will have 112 added to it (the X

coordinate will remain unchanged, since there is a 0 in that position). The next line has "("left/middle/right" (112 0)))". This specifies the column names and offsets. The "directions" statement indicates the directions linking each position ("n" for north, "e" for "east" and so on). The numbers after these names indicated which way to step for that direction. Use 0 for no change, or "1" or "-1" to got forward or backward. When Zillions reads this "grid" statement, then it will combine all this information to make a three by three grid, with names of positions like "top-left" and "bottom-right".

OK, we have a board now, what do we use to specify a piece? Why, the "piece" statement, of course:

Pieces

```
(piece
  (name man)
  (help "Man: drops on any empty square")
  (image X "images\TicTacToe\TTTX.bmp"
   O "images\TicTacToe\TTTO.bmp")
  (drops ((verify empty?) add))
)
```

The "name" section gives this piece a name: "man". The "help" section gives the text which Zillions will automatically display in the Status bar when the user points to the piece. You should put information about how the piece moves in this section. The "image" section gives the bitmap names for Zillions to use for each piece and for each player. You can make new piece bitmaps using a graphics editor like Windows Paint. Note that any fully green part of the bitmap will appear as "transparent". The "drops" section tells Zillions that this piece is dropped onto the board when it moves. The "(verify empty?)" section tells Zillions to make sure a position is empty before "add"ing it to the board.

Next, we tell Zillions how many pieces there are, and where they are located at the start of the game:

```
(board-setup
  (X (man off 5))
  (O (man off 5))
)
```

The "board-setup" section tells Zillions that there are 5 men for each player off the board at the start of the game.

Goals

OK, just one more section. How do we tell Zillions the goal(s) of the game? Easy, use "-condition" statements:

```
(draw-condition (X O) stalemated)
(win-condition (X O)
   (or (relative-config man n man n man)
      (relative-config man e man e man)
```

```
        (relative-config man ne man ne man)
        (relative-config man nw man nw man)
    )
  )
```

The "draw-condition" statement tells us that if any side is "stalemated" (has no legal moves left), then the game is a draw. The "win-condition" statement is a little more complex, so let's go over it:

After "(win-condition" we see "(X O)", the names of the players which this condition applies to. Next comes an "(or". This tells Zillions that any of the following conditions indicates a win. In Tic-Tac-Toe, you can win by placing three of your men in a row. They can be vertical (n direction), horizontal (e direction), or along any of the two diagonals (ne and nw directions). The "relative-config" statements tell Zillions that any position where a "man" piece is north of another "man" piece which is north of a third "man" piece indicates a win. This line is repeated for the other three directions.

Thats it. Zillions supplies lots more commands and ways to generate moves, add sound effects and music to your games, and detect more complex winning goals, but you have the basics now. Look through the keywords section to learn more about these and other Zillions commands. And remember, there are a host of examples to look at included with your copy of Zillions of Games.

Creating Variants

Zillions of Games lets you create game variants with very little work. Lets say you wanted to create a variant of Tic-Tac-Toe where the first player to get three-in-a-row loses. The only changes you would need are in the game title, description and the goal of the game. Just add these lines to the sample ZRF file:

```
(variant
  (title "Losing Tic-Tac-Toe")
  (description "This is the same as normal TicTacToe, except that    the
     object is NOT to get 3-in-a-row.\\
     One side takes X's and the other side takes O's. Players alternate
     placing their marks on open spots. The object is to avoid getting three    (
     your marks in a row horizontally, vertically, or diagonally. If    there are
     no 3-in-a-rows, it's a cat's game (a draw).")
  (loss-condition (X O)
    (or (relative-config man n man n man)
        (relative-config man e man e man)
        (relative-config man ne man ne man)
        (relative-config man nw man nw man)
    )
  )
)
```

You can see that we have changed the title and description to better describe the new variant, plus the win-condition has been changed to a loss-condition.

Zillions will use the other sections from the main game description (like the board, pieces and so on) and just substitute the changed sections (title, description and loss-condition in this case). Simple

variants can be created in a few minutes.

Moves Example: Defining a Cannons Movement

Moves Example: Defining a Cannons Movement

This example will take you step by step, explaining how the moves definition of a Chinese Chess Cannon piece is put together. First, notice this line near the top of Chess,_Chinese.zrf:

```
(define slide-cannon ($1 (while empty? add $1) $1 (while empty? $1) (verify    e
```

This line is used later on, in the definition of the Cannon. In fact, if you look at the Cannon piece definition, you will see these lines:

```
(moves
  (slide-cannon n)
  (slide-cannon e)
  (slide-cannon s)
  (slide-cannon w)
)
```

Zillions will read in this and substitute the value after "slide-cannon" for the holder "$1", like this:

```
(moves
  (n (while empty? add n) n (while empty? n) (verify enemy?) add)
  (e (while empty? add e) e (while empty? e) (verify enemy?) add)
  ;...and so on
)
```

You see that the macro of "slide-cannon" just makes it easier for us to write the same code for different directions, saving the game writer (you) time.

The basic thing to keep in mind with move generation code is that it is just a series of steps that tell the computer how to generate the moves, testing whatever is appropriate as it goes. Once Zillions knows enough about a move, the add command tells Zillions to write out that move, so it can search it later.

Now the Cannon in Chinese Chess can slide like a rook until it hits any piece. It can't capture that piece, but it can jump over that piece and capture another enemy piece behind it.

Lets step through the instructions now. I'll use the north direction as an example. Okay, first we have:

```
(moves
```

moves tells Zillions that the Cannon is a moving piece, meaning it moves from one square to another. Pieces can also be drops, meaning they move from off the board onto the board.

Next, we see:

```
(n
```

This just starts off at the Cannon's current location and steps one square to the north. Next:

```
(while empty? add n)
```

This is a while statement. The "while" statement will keep repeating something as long as the condition right after the "while" command is still true. In this case, as long as the current square is empty (the empty? part), the loop part of "add n" will be repeated.

The "add n" part does this: First, the add writes out the current move. The "from" position for the Cannon is simply where it is on the board right now (this is implied in the move generation). The "to" part of the move is the current location. So what this will do is add a move for each empty square in the north direction, until the square is no longer "empty?". Whenever the keyword "add" is used, the current "from" and "to" positions are written out as a move.

The next statement is simply "n", so the moves "to" position once again one square to the north, in this case to the empty square just past the position the previous (while empty? ...) statement found. The next part is familiar:

```
(while empty? n)
```

You notice that there is not "add" this time around, since the only valid move for a Chinese Chess Cannon to make after "jumping" over another piece is a capture. So this code piece just looks for a non-empty square past the piece that was just jumped over.

Only one more section to go:

```
(verify enemy?) add)
```

This does two things. The "(verify enemy?)" part just makes sure that the piece which was found is the enemy (we wouldn't want to capture our own piece, since that is illegal in Chinese Chess). If the piece is not an enemy, then move generation stops. Otherwise, the current move is the "add"ed.

The final ")" just balances things out, and tells Zillions that move generation for this direction is finished.

The Internals of Movement

We define a move primitive to be a basic element of a generated move that changes the position in some way, such as the flipping of single piece in a Reversi move. The following sections describe the internals of movement: how move primitives are written out when generating a move, how they executed in a played move, how they affect move entry and animation, and when data defined in the move code is cleared.

What It Is To Generate Moves

It is important to keep in mind the difference between generating a move and playing it out on the board. In a moves or drops block you are setting down the logic to generate moves. Moves will be generated whenever Zillions needs a list of moves in a given position. When a move is generated by an add it is placed on list of legal moves. It is not actually played on the board at that time and it may in fact never be played.

This is important because inside your move generation logic you will be querying the state of the board, e.g. is the current position occupied?, and you must realize that changes in your move logic will not have been applied at this point. A capture you are making as part of the move, for example, will not have taken place yet. The move that is being generated as legal may very well never be played out, in which case the capture will never occur.

The Order Move Primitives Are Played Out Within A Move

When an add, add-copy, add-partial, or add-copy-partial is encountered when generating moves, the move primitives are written in the following order (from first to last):

1. add, add-copy, add-partial, add-copy-partial, cascade
2. change-type, create, flip, change-owner, set-attribute
3. capture

They are executed internally in the order they are written out. This means that when the move is actually played out on the board or in the search, first come the movements (including promotions and capturing as a result of the movement), then all modifications of pieces, and finally all captures with the capture keyword. Within a level, the order is not guaranteed, e.g. you should not count on a move being able to create a piece and then flip it.

Note that if the cascade keyword is used, defining multiple piece movements within a single move, all of these movements are executed simultaneously. First all the pieces are picked up, then all landed-on pieces are removed, and then the pieces are set down again. This allows you to rotate a set of pieces (see the Rotate 9 variant of the 15 Puzzle) without having to worry that pieces you are moving will be captured.

The Key Move Primitive

The first move primitive in the move is defined as the key move primitive. For example, in Chess the King movement in castling is added first with a cascade and is therefore the key move primitive. The Rook move is added after the cascade. If there are no cascades, the key move primitive will be the add.

When determining if a user's click matches a move, only the key primitive part of the move is matched. The key move is animated first. Then all the other move primitives are animated in the order they appear.

When Move Data Is Reset

In move generation Zillions keeps track of current states and builds up lists of capture's and other actions to be written out as part of a move when an add command is encountered. These states and lists are cleared or reset at various times during move generation.

The following are reset after a move is added (with add or one of the its variants) in move generation:

- all change-types, flips, create's, change-owners, set-attributes, and captures
- all cascade's, but only if the "discard cascades" option is set to true

The following are reset after each cascade or add (including add- variants):

- from, to

A "move generation block," or simply "move block," is a parenthesized section occurring directly within a moves or drops construct. At the beginning of a move generation block, the following are reset:

- mark positions
- position flags
- all cascade's (if "discard cascades" option is set to its default value of false; otherwise cascade's will be discarded following every add.)

Randomness

Zillions supports randomness in games through a "random player", a special, computer-controlled player that chooses its move randomly. Random players can be used in many clever ways. For example, though dice arent directly supported in Zillions, a random player can be made to simulate dice. You can see this in Senat.zrf, where the "?Dice-Toss" player is a random player whose moves are different rolls of a die. Each turn, it arbitrarily chooses between its "die moves", effectively rolling a die in the game. Then the move logic for the next player can be based on which die was rolled. Multiple dice can be simulated by consecutive moves of random players. To denote a random player, name the player starting with a question mark "?".

TechNote #1: How To Write An Engine Plug-In For Zillions of Games

By Jeff Mallett - Copyright 2000 Zillions Development Corporation
Revised 12/1/00

The following technical documentation is intended for third-party developers interested in programming game-specific engines to replace Zillions of Games' default engine. It assumes fluency in programming and is not intended as end-user documentation.

Zillions of Games allows the use of plug-in engines, add-on game AIs hard-coded to come up with moves for specific games. These exist in the form of DLLs that Zillions calls according to a predefined interface. As DLLs they can be written in any language and compiled in a variety of development environments.

The Zillions of Games CD-ROM comes with two engine plug-ins, which handle certain variants from the Go-Moku and Reversi ZRFs. You'll find them in: **Zillions/Engines/**

A ZRF indicates that it wants to use a plug-in by including an engine command. For example,

```
(engine "Engines\Jello.dll")
```

indicates the use of the plug-ing called "Jello.dll" in the Engines directory.

Engine Routines

There are only four routines that the plug-in engine needs to support. These are defined in the following C header files. The routines allow Zillions to tell the engine to start a new game, make a move in the real game, search from the current position, and clean up. There are also two optional routines which allow the engine to generate moves for a game and determine the result of the game.

The engine returns a **DLL_*** constant back to the Zillions, which should be **DLL_OK** under normal circumstances. If the engine returns a negative error code, Zillions of Games will report this to the user and then unload the engine plug-in. If an engine plug-in is unloaded, either for this reason or because it returned a move that Zillions did not recognize as valid, Zillions will revert to using its built-in, universal engine.

```
// EngineDLL.h
//
// Copyright 1998-2000 Zillions Development
//
// Shared DLL plug-in for DLL engine and Zillions

#include "windows.h"

typedef enum {
    kKEEPSEARCHING = 0,
    kSTOPSOON = 1,
    kSTOPNOW = 2
} Search_Status;
```

```
typedef enum {
    DLL_OK = 0,
    DLL_OK_DONT_SEND_SETUP = 1, // only supported in 1.0.2 and higher!

    DLL_GENERIC_ERROR = -1,
    DLL_OUT_OF_MEMORY_ERROR = -2,
    DLL_UNKNOWN_VARIANT_ERROR = -3,
    DLL_UNKNOWN_PLAYER_ERROR = -4,
    DLL_UNKNOWN_PIECE_ERROR = -5,
    DLL_WRONG_SIDE_TO_MOVE_ERROR = -6,
    DLL_INVALID_POSITION_ERROR = -7,
    DLL_NO_MOVES = -8
} DLL_Result;

enum {
    UNKNOWN_SCORE = -2140000000L,
    LOSS_SCORE = -2130000000L,
    DRAW_SCORE = 0,
    WIN_SCORE = 2130000000L
};

// ***** REQUIRED ROUTINES

// DLL_Search
//
// The DLL should search from the current position. If it returns DLL_OK it should
// also return the best move found in str; however, it should not make the move
// internally. A separate call to MakeAMove() will follow to make the move the
// engine returns.
//
// -> lSearchTime: Target search time in milliseconds
// -> lDepthLimit: Maximum moves deep the engine should search
// -> lVariety: Variety setting for engine. 0 = no variety, 10 = most variety
// -> pSearchStatus: Pointer to variable where Zillions will report search status
// -> bestMove: Pointer to a string where engine can report the best move found so far
// -> currentMove: Pointer to a string where engine can report the move being searched
// -> plNodes: Pointer to a long where engine can report # of positions searched so far
// -> plScore: Pointer to a long where engine can report current best score in search
// -> plDepth: Pointer to a long where engine can report current search depth
//
// Returns DLL_OK or a negative error code

typedef DLL_Result (FAR PASCAL *SEARCH)(long lSearchTime, long lDepthLimit, long lVariety,
    const Search_Status *pSearchStatus, LPSTR bestMove, LPSTR currentMove,
    long *plNodes, long *plScore, long *plDepth);


// DLL_MakeAMove
//
// The DLL should try to make the given move internally.
//
// -> move: notation for the move that the engine should make
//
// Returns DLL_OK or a negative error code

typedef DLL_Result (FAR PASCAL *MAKEAMOVE)(LPCSTR move);
```

```
// DLL_StartNewGame
//
// The DLL should reset the board for a new game.
//
// -> variant: The variant to be played as it appears in the variant menu
//
// Returns DLL_OK, DLL_OK_DONT_SEND_SETUP, DLL_OUT_OF_MEMORY_ERROR, or
//   DLL_GENERIC_ERROR

typedef DLL_Result (FAR PASCAL *STARTNEWGAME)(LPCSTR variant);

// DLL_CleanUp
//
// The DLL should free memory and prepare to be unloaded.
//
// Returns DLL_OK, DLL_OUT_OF_MEMORY_ERROR, or DLL_GENERIC_ERROR

typedef DLL_Result (FAR PASCAL *CLEANUP)(void);


// ***** OPTIONAL ROUTINES

// DLL_IsGameOver
//
// This optional function is called by Zillions to see if a game is over.  If
// not present, Zillions uses the goal in the ZRF to decide the winner.
//
// -> lResult: Pointer to the game result which the DLL should fill in when
//              called.  If the game is over the routine should fill in WIN_SCORE,
//              DRAW_SCORE, or LOSS_SCORE.  Otherwise the routine should fill in
//              UNKNOWN_SCORE.
// -> zcomment: Pointer to a 500-char string in Zillions which the DLL can optionally
//              fill in, to make an announcement about why the game is over, such
//              as "Draw by third repetition".  The DLL should not modify this
//              string if there is nothing to report.
//
// Returns DLL_OK or a negative error code

typedef DLL_Result (FAR PASCAL *ISGAMEOVER)(long *lResult, LPSTR zcomment);


// DLL_GenerateMoves
//
// You can use GenerateMoves in your DLL to tell Zillions the legal moves for
// any position in the game.
//
// -> moveBuffer: Pointer to a 1024-char sting which the DLL should fill in when
//              called.  Initial call should be with moveBuffer set to "".  Each call
//              to GenerateMoves should fill in the next available move from the
//              current position, with a final "" when no more moves are available.
//              All moves must be in valid Zillions move string format.
//
// Returns DLL_OK or a negative error code

typedef DLL_Result (FAR PASCAL *GENERATEMOVES)(LPCSTR moveBuffer);
```

```
// Engine.h
//
// Copyright 1998-2000 Zillions Development
//
// Header file for plug-in DLL engine to Zillions
```

```
#include "EngineDLL.h"

DLL_Result FAR PASCAL DLL_Search(long lSearchTime, long lDepthLimit, long lVariety,
    Search_Status *pSearchStatus, LPSTR bestMove, LPSTR currentMove,
    long *plNodes, long *plScore, long *plDepth);
DLL_Result FAR PASCAL DLL_MakeAMove(LPCSTR move);
DLL_Result FAR PASCAL DLL_StartNewGame(LPCSTR variant);
DLL_Result FAR PASCAL DLL_CleanUp();
DLL_Result FAR PASCAL DLL_IsGameOver(long *lResult, LPSTR zcomment);
DLL_Result FAR PASCAL DLL_GenerateMoves(LPCSTR moveBuffer);
```

The engine does not call Zillions of Games. However, during a search it can find out from Zillions whether it should continue searching. When **DLL_Search** is called, the engine should store away the argument **pSearchStatus** and then refer to it during the search. If the user requests that the program move now or the time has expired, the value will change to **kSTOPSOON**. In this case the engine should return a result as soon as possible. If Zillions of Games needs to abort the search prematurely, e.g. the user has chosen to exit the program, the value will change to **kSTOPNOW**. In this case the engine should return as soon as possible, whether or not a good result is available. The engine should not change the value of ***pSearchStatus** itself.

During a search the engine should continually report its own search status by updating the values of ***plNodes**, ***plScore**, and ***plDepth**. Zillions uses these values to display feedback on progress to the user. In order to display this feedback during the search, the engine needs to periodically give Zillions a chance to process Windows messages. For example, in C/C++ you might have the following code:

```
MSG msg;
while (PeekMessage(&msg;, NULL, 0, 0, PM_REMOVE)) {
   TranslateMessage(&msg;);
   DispatchMessage(&msg;);
}
```

***plScore** is assumed to be the following units:

```
 -2,140,000,000 (UNKNOWN_SCORE): Score isn't known
    -2,130,000,000 (LOSS_SCORE): Immediate loss
-2,130,000,000 (LOSS_SCORE) + x: Loss in x (partial) moves
    (-2,130,000,000+150)..-1001: Bad for side on the move
                  -1000...1000: Close game
      1001..(2,130,000,000-150): Good for side on the move
  2,130,000,000 (WIN_SCORE) - x: Win in x (partial) moves
```

As a DLL, the plug-in will also need to have the following two routines defined. The **LibMain** function will be called when the DLL is loaded, so that gives you a chance to do once-only initialization. (Another method is to simply check a static variable inside **DLL_StartNewGame**.) The following code gives an example implementation of these two routines that does nothing except initialize the C library's randomizer.

```
#include <time.h>
int FAR PASCAL LibMain(HANDLE hInstance, WORD wDataSeg, WORD wHeapSize)
```

```
{
    srand( (unsigned)time( NULL ) );
    return 1;
}

int FAR PASCAL WEP(void)
{
    return 1;
}
```

With these functions the **EXPORTS** section of your project's .DEF file might look like this (the last two routines are optional):

```
EXPORTS
   LibMain
   WEP
   DLL_Search
   DLL_MakeAMove
   DLL_StartNewGame
   DLL_CleanUp
   DLL_IsGameOver
   DLL_GenerateMoves
```

Passing Moves and Edits

Moves are passed back and forth as move strings. These move strings are the same as those written out to a saved game, such as "Pawn e2 - e4", and don't include the move number.

For most games these strings are also the same as the move strings displayed in the move list (the part following the move number), which makes it easy for you to see what is being passed to your engine. There are two exceptions:

1) The move involves setting piece attributes. For example, when a Rook in Chess moves, the ZRF sets an attribute saying that the Rook may no longer be used in castling. The setting of piece attributes is not displayed to the user on the movelist, but it is sent to the engine in case the engine needs that information in differentiating moves. Example: "Rook h1 - f1 @ f1 0 0"
2) The move is a partial move. In this case a "partial" string is added. Example: "partial 2 Checker d4 - b6 x c5"

To see the exact format of moves, simply look at what is saved in the .ZSG.

Note that the sending moves to the engine serves two purposes, both as a method to pass actual game moves and to pass edits made to the board. Thus, your engine needs to handle any edits the user can make to the board, for instance "(White Rook e4)". The Zillions of Games GUI and engine is flexible enough to deal with any position resulting from edits. If your engine can't cope with the existing board position, for example, it is a Chess program that is hard-coded to assume that there is exactly one White King and one Black King, then the engine can indicate this by returning the code **DLL_INVALID_POSITION_ERROR**.

Edits to the board are passed just as they are displayed in the move list, as a capture or drop surrounded by parentheses.

After calling **DLL_StartNewGame** Zillions of Games will always pass down a series of board edits to place all the initial pieces on the board. This was done for two reasons:

- It allows the engine to be simpler. The engine can simply assume at the start of the game that the board is empty and let Zillions handle the logic for setting up the board correctly.
- It allows an engine to be used in a wide variety of games without re-coding. Many games, such as Blobs or Turn Off, can be played with the same rules and board, but with different starting configurations. For example, someone could write a ZRF variant that uses an existing engine in new setups. Also, at some point Zillions of Games may support setup randomization, in which case, an existing Chess engine could be used to play Shuffle or Fischer Random chess.

Of course, your engine may prefer a certain setup, for instance, if it has an opening book. If getting the series of initial board edits is a problem, your **DLL_StartNewGame** should return **DLL_OK_DONT_SEND_SETUP** to indicate you're assuming a fixed setup and want to bypass this stage.

Graphics

Zillions requires that all piece and board graphics be in .BMP format. Any editing program will do for editing BMPs. For simple graphics, you can use MS Paint (Programs:Accessories:Paint) or shareware programs on the web like PaintShop Pro.

Zillions can read in standard BMP files of any bit depth. We recommend using 256 color bitmaps for the boards and full 24 bit bitmaps for the pieces.

Though BMPs are rectangular, most pieces will not be perfectly rectangular and you値l need to make the space around the edges of the piece transparent. Zillions treats pure green (0, 255, 0) as transparent when drawing bitmaps.

Graphics files are specified through the image statement.

Advanced Topic FAQ

Q: What is the default for repetition checking?

Zillions doesnt think it makes sense in any game to endlessly repeat moves, so it checks to see if the position has been repeated three times. If so, the default is to declare a draw. This check does not occur in puzzles, since a player may need to repeat the same position many times until he hits on the solution.

Q: What are the limitations in Zillions for a ZRF?

Maximum number of players in a game: **32**
Maximum number of pieces in a game: **no limit**
Maximum number of positions in a game: **no limit**
Maximum number of uniquely named piece attributes in a game: **32**
Maximum number of uniquely named zones in a game: **32**
Maximum number of game-defined directions: **100** *(64 prior to v2.0)*
Maximum number of dimensions in a grid statement: **5**
Maximum number of flags: **32** *(11 prior to v1.0.3)*
Maximum number of positional flags: **16**
Maximum size of compiled move code in a move block of a piece: **32k** *(10k prior to v.1.2; 20k prior to v.2.0)*
Maximum number of pieces a single move/drop may capture: **no limit** *(256 prior to v.2.0; 32 prior to v.1.0.2)*
Maximum number of cascades a single move/drop can perform: **1024** *(256 prior to v.2.0; 32 prior to v.1.0.2)*
Maximum number of flips, change-owners, change-types, and set-attributes a single move/drop can do: **no limit** *(256 prior to v.2.0; 32 prior to v.1.0.2)*
Maximum unique symbols in a ZRF: **no limit** *(1,000 prior to v1.1.1)*
Maximum number of macro definitions: **no limit**
Maximum number of macro arguments: **no limit**
Maximum amount of text in a single string or expanded macro: **256k** (or less if RAM low) *(128k prior to v.2.0)*
Maximum number of move primitives that the search tree can contain: **20,000** *(16,000 prior to v.2.0; 12,000 prior to v1.0.3)*
Maximum number of instructions that a move/drop can execute: **200,000** *(32,000 prior to v.1.2; 99,999 prior to v.2.0)*

Q: Is there a way to find out inside a move definition which player's turn it is?

Yes, though not through a direct call. One way is to use zones, taking advantage of the fact that they are relative to the side that is moving. Look at these macros from the implementation of Mini-Shogi:

```
 (define am-white? (not-in-zone? promotion-zone 1i))
       (define am-black? (in-zone? promotion-zone 1i))
       ...
       (zone
         (name promotion-zone)
         (players Black)
           (positions 5i 4i 3i 2i 1i)
       )
       (zone
         (name promotion-zone)
         (players White)
         (positions 5v 4v 3v 2v 1v)
       )
)
```

If you have no zones defined you can define a small one, with one square in it for each side.

Q: Is there a way to find out inside a move definition which variant you are in?

Not directly, though there are lots of ways you can handle this indirectly. For example, you can create a dummy position "v" and, in the board setup, place a different dummy piece there in every variant, say pieces "Variant1", "Variant2" and so on. Then your move code can check to see if youre in the first variant like so: `(piece? Variant1 v)`

Q: What happens if I go off the edge of the board in my move logic?

A move-block (one of the lists inside moves statement) is terminated whenever the current position becomes invalid, such as going off the edge of the board. The move generator assumes that once the position is invalid, the rest of the move generation for that move won't make any sense. This prevents some common endless loops, for instance,

```
(while true n)
```

and also makes things more efficient.
If you want to check the contents of adjacent positions without the risk of prematurely ending the generation logic for this move, use a syntax like:

```
 (piece? Knight sw)
```

rather than

```
 sw (piece? Knight) ne
```

The former piece of code tests the position without changing the current position. This is okay and

will simply return false if there is no position in the sw direction. The second code fragment actually moves the current position in the direction sw, which will stop the processing if the position doesnt exist.
Another possibility is to use the "on-board?" test. For example,

```
(if (on-board? sw) sw (piece? Knight)
```

Q: If a player's move is multiple captures, then is

"(loss-condition (opponent) (pieces-remaining n opponent))"

checked after each jump, or only at the end of the player's move?

All win/draw/loss conditions are tested after a whole move is made. Checking is not done after a partial-move, such as a jump in Checkers, unless it is the final jump of the series.

Q: Is there a simple way to use "grid" and "kill-positions", "links", and "unlink" to end up with a hex board?
grid is most useful as a shortcut making rectangular boards. There isnt any direct language support for hex boards yet. Please look to Chinese_Checkers.zrf for an example of how to make a hex board. The positions are really hexagons.

Q:What does last-from? return after a drops move?
Nothing. But last-to? will be the position the piece was dropped to.

Q: move-priorities: Can I specify moves which have the same priority, as in "move-priorities move1 (move2 move3 move4) move5 (move6 move7)"?
Nope. It is possible to assign the same move-type to multiple moves of multiple pieces though. For example, the moves of the knight and bishop might both be assigned (move-type minor-piece)

Q: Can you give any hints for speeding up my ZRF?

1) Avoid unnecessary move code. Often move code can be tightened up or lines are simply

redundant. For example, an explicit "to" is usually not necessary right before adding a move. The less logic Zillions has to process to generate a move, the better.
2) Avoid using position flags if possible. The position flag for each position must be initialized for each move block for every piece. Especially when there are lots of positions, this can be very time-consuming. If your game doesn't use position flags, Zillions can skip this step.
3) Use zones when possible. Using zones allows your game to test against a range of positions in a single shot rather than having to iterate over each possibility.
4) Use the alternate forms of <drop-def> when only dropping to part of a board. For example, this:

```
(moves (my-zone ... ))
```

is much faster than:

```
(moves ( (verify (in-zone? my-zone)) ... )
```

In the former case, Zillions only looks at positions in my-zone. In the latter, the move logic is evaluated for *every* position on the board (which also means extra initializing of position flags, etc.)
5) Consider carefully how the positions are linked together. For example, if you have a piece that makes big hops, normally your move logic would step through the intervening positions one by one, checking for the board edge, until it arrives at the destination of the hop. However, it is probably quicker to create a set of new directions just for that piece that directly link positions it can hop between.

Q: Which is better, using the syntax "in-zone? red", or using "set-position-flag red" then querying using the syntax "red?".

Definitely (in-zone? red)
1) zones are a static part of the game definition, whereas position flags have a short lifetime, only existing during a move generation for one of the kinds of moves of a piece. To make it work with position flags, you'd have to set them every time you generated a new kind of move for a piece. Zones are initialized only once, when the ZRF is read in.
2) If any position flags are used in a game, the position flags are initialized to false at the start of every type of move generation. So there's a big performance penalty for using position flags. This initialization is bypassed when a game/variant doesn't use them.

Q: The initial selection screens are implemented as games, but act differently. Is it possible for me to make my own selection screen(s)?

Yes. In a selection screen board editing is disabled along with many features normally available from the menus. Detection of the end of the game is turned off and toolbars are hidden. At present, all you need to do to turn a variant into a selection screen is to use the open command somewhere in

the variant.

Q: I made a game where players drop pieces on the board, but when I try it, Zillions says it's a stalemate. What's wrong?

Even though you've defined the drop moves, unless the game starts with pieces off the board, they'll be nothing for the players to drop. You need to use the "off" command in the board-setup to tell Zillions that there are pieces to drop on the board.

Can't find your question here? Need a guiding hand? We invite you to the Zillions Discussion Boards on the web, where game authors get together to discuss all angles of game design and implementation.

;

A semi-colon indicates a note or comment: everything following on the line is ignored. Annotating a ZRF with comments makes it easier to understand.

Examples

```
(verify empty?) ; Make sure there is nothing in the way
```

define

define specifies a macro that can be called.

```
(define <macro-name> <macro-definition>)
```

Remarks

Macros can save you time and make your rules easier to maintain by letting you repeat commonly used sections. Calls take the form of the macro name surrounded by parentheses. The body of a macro is directly substituted for the calls during before the rules are parsed.

Before pre-processing:

```
(define <macro-name> <macro-definition>)
...
(<macro-name>)
...
(<macro-name>)
```

After pre-processing:

```
...
<macro-definition>
...
<macro-definition>
```

Macros may be defined before or after they are used.

Arguments

If desired, arguments can be used in macro call. In the macro definition, $1 stands for the first argument, $2 for the second argument, and so on. So if you define a macro called "blah":

```
(define blah n $1 n $1 $2)
```

then this macro call:

```
(blah sw se)
```

will get replaced by this:

```
n sw n sw se
```

The first argument (sw) got directly substituted in for $1 and the second argument (se) got substituted in for $2. In C you would achieve the same thing with a syntax like this:

```
 #define blah(s1, s2) n s1 n s1 s2
```

Usage

```
(define east
  (opposite west)
)
```

Examples

```
; Macro definition
; Go in direction $1 until hit obstacle or edge of board
(define slide (while (empty? $1) $1) )

...

(piece
 (moves ; Pieces move along straight liens as far as possible
   ( (slide n) add) ; Macro call
   ( (slide s) add) ; Macro call
   ( (slide e) add) ; Macro call
   ( (slide w) add) ; Macro call
 )

; *** THE ABOVE CODE EXPANDS INTERNALLY TO THE FOLLOWING:

(piece
  (moves ; Pieces move along straight liens as far as possible
    ( (while (empty? n) n) add) ; Macro call
    ( (while (empty? s) s) add) ; Macro call
    ( (while (empty? e) e) add) ; Macro call
    ( (while (empty? w) w) add) ; Macro call
  )
)
```

See Also

include

include

Inserts ZRF code from a different file at the location before code is parsed.

```
(include <file-name-string>)
```

Remarks

While not used in any ZRFs on the CD-ROM, include can be useful when you want to include a set of rules code in multiple ZRFs. The contents of the file are inserted at the spot, like the substitution of a macro. Include may be used anywhere with a ZRF.

Usage

```
(include "MyLibrary.txt") ; Replace this call with contents of MyLibrary.txt
```

Examples

None.

See Also

define

version

Identifies a ZRF with a particular version of Zillions.

```
(version <version-string>)
```

Remarks

If a ZRF includes a **version** statement, then Zillions will check the version string against the version of Zillions of Games. If Zillions knows about this version it will read it in; otherwise, it will display a message to the user that the ZRF requires a newer version of Zillions

If a ZRF does not include a **version** statement, then Zillions will always try to read it in.

Currently known version strings are "1.0", "1.2", "1.3", "2.0"... For example, all versions of Zillions recognize the version string "1.0", whereas only versions 1.2 or higher recognize the "1.2" string. See the About box for version number. You can specify known maintenance versions, such as "2.0.1", but don't include any letters in the version, e.g. "2.0p" is not allowed.

Using version strings avoids having users see parse errors and therefore is recommended if you know that the ZRF relies on newer language features. However, your ZRF should probably not require a higher version number than is actually necessary to open a ZRF. If your ZRF uses (go last-from), or (go last-to) your ZRF should include a (**version** "1.2") or higher statement. If your ZRF uses the option statement, you should at least include (**version** "2.0").

Like game and variant statements, **version** statements should be at the top level, not be embedded inside any other constructs. We recommend that you put it first in the ZRF, so it is easy to find. Because of its nature, **version** is detected very early in game parsing and so shouldn't be a product of a macro or an include. Finally, there should never be more than one **version** statement per ZRF.

Usage

```
(version "1.2") ; requires version 1.2 or higher
```

Examples

None.

See Also

option go game, variant define include

<condition>

An expression that evaluates to a Boolean value.

```
true
false
(flag? <flag-name>)
(not-flag? <flag-name>)
<position-condition>
(not <condition>)
(and <condition>...<condition>)
(or <condition>...<condition>)
```

Remarks

Each of these conditions evaluates to a True or False as follows:

true	Always evaluates to True
false	Always evaluates to False
flag?	Returns the current True/False value of the flag set with set-flag
not-flag?	Returns True if the flag is set to False, or False if the flag is set to True
(not...	Returns True if the condition is False, returns False if the condition is True
(and...	Returns True only if all the conditions within it return True *Note: if a condition returns False, the remaining conditions will not be looked at*
(or...	Returns True if any of the conditions within it return True *Note: if a condition returns True, the remaining conditions will not be looked at*

Conditions can be used in a variety of places, such as in if , set-flag, and verify constructs.

Usage

```
(game
...
  (piece
    ...
    (moves
    ...
      (...(verify false)...)
      (...(set-flag my-flag true)...)
      (...(if (flag? my-flag)...)
    ...
    )
  ...
  )
...
)
```

Examples

None.

See Also

if set-flag verify

<goal>

Checks to see if the game is over.

```
stalemated
repetition
(captured <piece-type>...<piece-type>
(checkmated <piece-type...<piece-type>
(absolute-config...)
(relative-config...)
(pieces-remaining...)
(total-piece-count...)
```

Remarks

A goal is a condition Zillions checks to see if the game is over. Zillions checks goals after an entire move is made, not in the middle of a series of partial moves.

The **not**, **and**, and **or** operator constructs use regular goals as their arguments.

```
(not <goal> )
(and <goal>...<goal> )
(or <goal>...<goal> )
```

Any number of regular goals may follow an **and** or **or** operator.

Top-level Goals

These must appear at the highest level, i.e. never within a **not**, **and** or **or** construct.

```
stalemated
repetition
(captured <piece-type>...<piece-type>)
(checkmated <piece-type>...<piece-type>)
```

Regular Goals

These may appear at any level.

```
(absolute-config ... )
(relative-config... )
(pieces-remaining... )
(total-piece-count... )
```

Click on the above links to see information about a particular goal.

Usage

```
(game
...
  (loss-condition (White Black)
    ; if either Black or White lose their King, they lose the game
    (captured King)
  )
...
)
```

Examples

```
; The game is a draw if White is stalemated
(draw-condition (White) stalemated)

; White wins if he has a Stone on d4 with 3 pieces on the board
(win-condition (White)
  (and
    (pieces-remaining 3)
    (absolute-config Stone (d4))
  )
)

; Its a draw if Black has no Tiles or if Black has a Stone
; on d4 with 3 on the board.
(draw-condition (Black)
  (or
    (pieces-remaining 0 Tile)
    (and
      (pieces-remaining 3)
      (absolute-config Stone (d4))
    )
  )
)
```

See Also

stalemated repetition captured checkmated total-piece-count absolute-config relative-config pieces-remaining

<instruction>

An instruction is a basic command in the move language. These are the forms for a move language instruction. "|" indicates a choice between arguments. Brackets indicate an optional section.

<direction>

<position>

add | add-copy | add-partial | add-copy-partial
(add | add-copy | add-partial | add-copy-partial <position> | <direction>)

back

capture
(capture <position> | <direction>)

cascade

change-owner
(change-owner <position> | <direction>)

(change-type <piece-type> [<position> | <direction>])

create
(create [<player>] [<piece-type>] [<position> | <direction>])

flip
(flip <position> | <direction>)

from

(go from | to | last-from | last-to | mark)

(if <condition> <instruction> ... <instruction> [else <instruction> ... <instruction>])

mark
(mark <position> | <direction>)

(opposite <direction>)

(set-attribute <attribute> <condition>)

(set-flag <flag-name> <condition>)

to

(verify <condition>)

(while <condition> <instruction> ... <instruction>)

<position-condition>

Is something about this position true?

```
<position-condition>
(<position-condition> <position>)
(<position-condition> <direction>)
```

Remarks

These are a set of expressions that evaluate to a Boolean (True or False) value based on a position. Some take an additional argument.

adjacent-to-enemy?	Is there a link from the position to a position with an enemy piece?
attacked? [<movetype>]	Is the position attacked by an enemy piece?
defended? [<movetype>]	Is the position defended by a friendly piece?
empty?	Is the position empty of pieces?
enemy?	Is there an enemy piece on the position?
friend?	Is there a friendly piece on the position?
goal-position?	Is this position in an absolute-config goal for any side? This includes positions in a zone used in an absolute-config goal.
in-zone? <zone>	Is this position in the given zone?
last-from?	Was the last move a move from this position?
last-to?	Was the last move a move to this position?
marked?	Is this the marked position?
neutral?	Is there a neutral piece on the position?
on-board?	Is this position defined? Use with a direction argument, i.e. (on-board? ne)
piece? <piece-type>	Is there a piece of this type on the position?
position? <position>	Is the current position the given position?
position-flag? <flag-name>	Is the given flag true for the current position?
<attribute>	Does a piece exist on the position that possesses the attribute?

For each <position-condition> with a built-in keyword, there is also an opposite version, which tests the negative condition:

not-adjacent-to-enemy?
not-attacked?

not-defended?
not-empty?
not-enemy?
not-friend?
not-goal-position?
not-in-zone? <zone>
not-last-from?
not-last-to?
not-marked?
not-neutral?
not-on-board?
not-piece? <piece-type>
not-position? <position>
not-position-flag? <flag-name>

This syntax is shorthand for using the not keyword, i.e. not-empty? is eqivalent to (not empty?)

A neutral piece is a piece owned by a neutral player, i.e. a player that has no moves in the turn-order. A friendly piece is any piece on the same side as the side that is moving. An enemy is any piece of a different side, even if it is neutral. At present, there is no concept of teams.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (ne (verify empty?) add)
    ...
    )
  ...
  )
...
)
```

Examples

```
empty? ; Is the position empty?
not-empty? ; Is the position occupied?
(empty? ne) ; Is the position in the "ne" direction empty?
```

```
(empty? e4) ; Is the e4 position empty?
(piece? Knight) ; Is there a Knight on the position?
(piece? Knight ne) ; Is there a Knight in the direction "ne" from the current pos
(piece? Knight e4) ; Is there a Knight on e4?
(position? e4) ; Are we at e4?
defended? ;Is this position being defended?

; Is the position in our zone called "my-promotion-zone"?
(in-zone? my-promotion-zone)

; Continue generating this move only if there is a neutral piece on the current p
(verify neutral?)

; Continue generating move only if there is NOT a neutral piece on the current po
(verify not-neutral?)

; If the position is empty then add a move to the current position
(if empty? add)

; Is the position in the "n" direction a valid position,
; i.e. is there a directional link "n" from the current position?
(on-board? n)

; Move the current position in the "n" direction until the edge of the board is r
(while (on-board? n) n)

; Set the flag to True if theres a friendly piece on the position, or False othe
(set-flag friend?)
```

See Also

attacked? defended? last-from? last-to? not-last-from? not-last-to? marked position-flag? not-position-flag? turn-order

<string>

```
"This is a string"
```

A string is a piece of literal text that may be displayed to the game player in some way.

Strings in zrf's (except in include statements) may be written over several lines. Single spaces are inserted between text on different lines. Blank lines are treated as carriage returns.

```
(description "foo line 1
  foo line 2

  foo line 4")
```

will be displayed as:

foo line 1 foo line 2
foo line 4

A backslash character ("\") is converted into a carriage return in the displayed text. Thus the above description could have also been written as:

```
 (description "foo line 1 foo line 2\foo line 4")
```

A backquote character ("`") is converted into a double-quotation-mark. For example:

```
 (description "foo `line 1` foo `line 2`\foo `line 4`")
```

will be displayed as:

foo "line 1" foo "line 2"
foo "line 4"

absolute-config

Stops the game if certain pieces are on certain positions.

```
(absolute-config <check-occupant>...<check-occupant> (<position-arg>...<position-
```

Remarks

The **absolute-config** goal evaluates to True if pieces are in certain configurations on the board. If the configurations are relative to other pieces and may be anywhere on the board, use relative-config instead.

<position-arg> may be one of the following:

<position>	Is there a piece at this position?
<zone>	Is there a piece on any one of these positions?

<check-occupant> may be one of the following:

<piece-type>	Do I have a piece of this type here?
(opponent <piece-type>)	Does someone else have a piece of this type here?
(any-owner <piece-type>)	Does anybody have a piece of this type here?

It may be negated in one of these ways:

```
(not <piece-type>)
(not (opponent <piece-type>))
(not (any-owner <piece-type>))
```

The check for a <position-arg> may be satisfied by any of the <check-occupant>s. In order for the goal to be true all of the <position-arg> checks must be satisfied.

It is possible to check in move code whether a position is part of an **absolute-config** goal by using the goal-position? keyword. The user can choose to highlight **absolute-config** goal positions by placing stars on them or the ZRF can force this using the "highlight goals" option.

Discussion

If you have

```
(absolute-config Knight (<arg1>))
```

it means that a Knight has to be in <arg1> for the goal to succeed. If this is a position:

```
(absolute-config Knight (a1))
```

then the Knight must be on position a1 for the goal to succeed. If it is a zone:

```
(absolute-config Knight (myzone))
```

and myzone has the positions a1 and b1 in it, then the Knight must be in either a1 OR b1. This is the difference between a zone and a position.

Now, if you have

```
(absolute-config Knight (<arg1> <arg2>))
```

it means that a Knight has to be in both <arg1> AND <arg2> for the goal to happen. So, if you have:

```
(absolute-config Knight (a1 b1))
```

then this means that a Knight must be on a1 AND b1. This is different (see above) than

```
(absolute-config Knight (myzone))
```

Now for the "not"s If you have this:

```
(absolute-config (not Knight) (a1 b1))
```

there must not be a Knight on a1 AND there must not be a Knight on b1. If you have this:

```
(absolute-config (not Knight) (myzone))
```

there must not be a Knight on a1 OR there must not be a Knight on b1. If either of these positions is empty the goal will succeed.

Usage

```
(game
...
  (win-condition (White Black)
    (absolute-config Pawn (a7 h7))
  )
...
)
```

Examples

```
; True if the side has a Pawn on a7
```

```
(absolute-config Pawn (a7))

; True if the side has a Pawn on a7 and a Pawn on h7
(absolute-config Pawn (a7 h7))

; True if the side has a Pawn or Knight on a7 and a
; Pawn or Knight on h7
(absolute-config Pawn Knight (a7 h7))

; True if the side has no Pawns on a7 and h7
(absolute-config (not Pawn) (a7 h7))

; True if the side has a Pawn on e4, or if anybody has a Bishop on e4
(absolute-config Pawn (any-owner Bishop) (e4))

; True if there are friendly Pawns on a7, h7, and somewhere in MyZone
(absolute-config Pawn (a7 h7 MyZone))

; True if at least one position in MyZone contains a friendly Pawn,
; Knight, or Bishop
(absolute-config Pawn Knight Bishop (MyZone))
```

See Also

relative-config any-owner goal-position? goal option

add, add-copy, add-partial, add-copy-partial

Add a move or drop to the list of legal moves.

```
add
(add <piece>...<piece>
add-copy
(add-copy <piece>...<piece>)
add-partial
(add-partial <piece>...<piece>)
(add-partial <move-type>)
(add-partial <piece>...<piece> <move-type>)
add-copy-partial
(add-copy-partial <piece>...<piece>)
(add-copy-partial <move-type>)
(add-copy-partial <piece>...<piece> <move-type>)
```

Remarks

These are the commands that actually cause a move to be generated. When an **add** is encountered all the information about the move is saved away, such as the positions. An **add** within a moves construct will generate a move of a piece from one square to another, while an **add** within a drops construct will generate the drop of a piece.

An **add** can also be used with one or more piece-type arguments, such as "(**add** queen)". This means that the piece turns into a queen. "(**add** queen rook bishop knight)" is used to define promotion in Chess. This syntax means the moving player may choose to promote to a queen, rook, bishop, or knight. Essentially four separate moves are generated with this one statement.

add-copy can be used just like **add**. Instead of adding a piece movement, it adds the duplication of a piece, thus creating a new piece on the board. It can be used with promotion syntax too, e.g. "**add-copy** queen rook bishop knight".

add-partial adds a "partial move" of a piece. After the partial move, the same piece is given a subsequent move if there is one available. An example of a partial-move is a capture in Checkers (Draughts). Unlike other moves, a partial-move does not change the side to move. In some respects a series of partial moves by a piece is considered one long move, e.g. the goal of a game is not checked until all the partial moves are complete. **add-partial** may also be used within a drops construct, allowing a piece to be dropped and then moved. **add-partial** can take an additional argument of a move type defined in a move-type construct. This means that only continuations of the given move type are possible. In the example of Checkers, capturing moves are the only possible continuations to a capturing move.

add-copy-partial is the addition of a partial-move that creates a duplicate of the moving piece (like **add-copy**).

More details about the internals of movement in Zillions are available.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (n add) ; Add a move of the piece one square to the north
    ...
    )
  ...
  )
...
)
```

Examples

None.

See Also

moves drops

any-owner

Use **any-owner** to determine if a piece is a certain type, regardless of the owner of that piece.

```
(any-owner <piece-type>)
```

Remarks

This is useful in games with neutral pieces.

Usage

```
(game
...
  (win-condition (White Black)
    (absolute-config (any-owner Neutron) (win-zone))
  )
  (win-condition (White Black)
    (relative-config (any-owner Counter) n (any-owner Counter))
  )
...
)
```

Examples

None.

See Also

absolute-config relative-config

attacked?

Determines whether a position is attacked by an enemy.

```
attacked?
(attacked? <position>)
(attacked? <direction>)
(attacked? <move-type>)
(attacked? <move-type> <position>)
(attacked? <move-type> <direction>)
not-attacked?
(not-attacked? <position>)
(not-attacked? <direction>)
(not-attacked? <move-type>)
(not-attacked? <move-type> <position>)
(not-attacked? <move-type> <direction>)
```

Remarks

attacked? returns true if the given position is attacked by an enemy piece, otherwise false. In checkmate games, a piece attacks a position even if moving to the position would put its side in check. To check attacks by a friendly piece use defended? instead.

Zillions determines whether a square is attacked by generating moves for the opponent to see if the opponent could take the piece on the position. If the position is currently vacant or occupied by another side's piece, Zillions will temporarily place a friendly piece there. In version 1.1.1 or later Zillions will place a piece of the same piece type that moves are being generated for. In earlier versions there is no guarantee as to what type of friendly piece will be placed.

When a move-type is passed in only attacks of the given move-type are examined.

not-attacked? returns false if the given position is attacked by an enemy piece, otherwise true.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
       (
       ...
         ; Verify that the current position is attacked by enemy
         (verify attacked?)
       ...
       )
    ...
```

```
    )
  ...
  )
...
)
```

Examples

```
(verify attacked?) ; Verify that the current square is attacked
(verify attacked? n) ; Verify that position in direction n is attacked
(verify not-attacked? e2) ; Verify that position e2 is not attacked

; Add a move only if the current square is attacked by enemy move
; of type jump-move
(if (attacked? jump-move) add)
```

See Also

defended?

attribute

Creates a new user piece attribute with an initial value of true or false.

```
(attribute <attribute> true)
(attribute <attribute> false)
```

Remarks

The **attribute** keyword is used to create a new user piece attribute. An attribute is simply a Boolean (true or false) variable attached to a specific piece on the board and carried with it throughout its lifetime. What the variable means is up to you. The true/false argument is the initial value of the attribute when a new piece enters the board, or at the start of the game.

For example, one kind of piece might be able to fire a laser-bolt once per game at another piece. You could define an attribute for this piece-type, such as:

```
(attribute lasers-charged? true)
```

to indicate the state of the laser beams. The variable is set to "true" to indicate the piece starts out charged. When a laser firing move is done, you would use:

```
(set-attribute lasers-charged? false)
```

to indicate that this piece can no longer fire lasers in this game. In your move logic you can test your attribute using the "lasers-charged" attribute as a <position-condition> , e.g.

```
(verify lasers-charged?)
```

You may have up to 32 differently named piece attributes in a game. More than one piece type can share the same attribute, e.g. there may be several piece-types that can fire lasers. Note that if a piece is changed during the game from one type of laser-firing piece to another with change-type, the "lasers-charged?" attribute will retain its value. If the new piece type has piece attributes that the old one didn't, or if they have different default values, you may want to call set-attribute yourself to initialize the values in the move changing the piece-type.

Usage

```
(game
...
  (piece
  ...
    (attribute strong-piece? true)
  ...
  )
...
```

)

Examples

None.

See Also

set-attribute

back, mark

Use the **mark** keyword to mark a position. **back** sets the current position to the marked position.

```
back
mark
(mark <position>)
(mark <direction>)
```

Remarks

If no positions have been marked then **back** sets current position back to the starting position for the move. Both **back** and **mark** are time savers, letting you return to a position quickly while generating moves.

You can test whether a position is marked or not using marked? and not-marked?

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      ( (while (empty? n) n)
        (verify enemy?)
        ; Mark the current position,
        ; which contains an enemy piece "n" of us
        mark
        ...
        back ; Go back to the marked position...
        add ; and add a move
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
mark ; Mark the current position
(mark e4) ; Mark e4
(mark ne) ; Mark the position in the "ne" direction from here
```

See Also

marked? not-marked?

board

Defines the playing board.

```
(board <board-arg>...<board-arg>)
```

Remarks

Boards are made of a number of parts, including an image bitmap, a number of positions which are linked together, and other features involving symmetry and zones. The arguments may be of one of the following forms:

```
(dummy <position><position>)
(grid <grid-arg>...<grid-arg>)
(image <file-string>)
(kill-positions <position>...<position>)
(links <direction> (<position>...<position>) ... (<position>...<position>) )
(positions <positions-arg>...<positions-arg>)
(symmetry <player> (<direction> <direction>) ... (<direction>...<direction>) )
(unlink <unlink-arg>...<unlink-arg>)
(zone <zone-arg>...<zone-arg>)
```

Usage

```
(game
...
  (board
  ...
  )
...
)
```

Examples

None.

See Also

dummy grid image kill-positions links positions symmetry unlink zone

board-setup

Defines where pieces start on the board at the beginning of the game.

```
(
  board-setup
  (<player> <placement>...<placement>)
  ...
  (<player> <placement>...<placement>)
)
```

Remarks

Each <placement> is of the form:

```
 (<piece-type> <position>...<position>)
```

This indicates that each position listed is filled with pieces of this type. The construct:

```
 off <number>
```

must be used in place of a to indicate how many of that kind of piece begin off the board, i.e. the player has that number of pieces to drop.

*WARNING! For games involving pieces which must be dropped on the board, dont forget to use the **off** command to tell Zillions how many pieces are available!*

Usage

```
(game
...
  (board-setup
    (Black (Disk off 99 d4 e5) )
    (White (Disk off 99 d5 e4) )
  )
...
)
```

Examples

```
(board-setup White (King e1) ) ; White starts with a King on e1
(board-setup Black (Disk d4 e5) ) ; Black starts with Disks on d4 and e5

; Black starts with Disks on d4 and e5, plus 99 off the board to drop
(board-setup Black (Disk off 99 d4 e5) )
```

```
(board-setup
  ; Black starts with Disks on d4 and e5, plus 99 to drop
  (Black (Disk off 99 d4 e5) )

  ; White starts with Disks on d5 and e4, plus 99 to drop  (White
  (Disk off 99 d5 e4) )
)
```

See Also

None.

capture

Captures a pieces on the given position.

```
capture
(capture <position>)
(capture <direction>)
```

Remarks

capture is used in a move definition to mark a piece for capture at a position other than the one your piece is moving to. Like create, change-type, flip, and change-owner, it affects the next move generated with an add. The position used is the current position, unless directed otherwise by an optional <position> or <direction> argument.

Usage

```
(game
...
   (piece
   ...
    (moves
    ...
     (n (capture n) (capture e) (capture w) add)
    ...
    )
  ...
  )
...
)
```

Examples

```
capture  ; Takes a piece on the current position
(capture e2)  ; Takes a piece on the position e2

; Takes a piece in the "n" direction
; This is the same as  (if (on-board? n) n capture s)
(capture n)
```

See Also

create change-owner change-type flip

captured

Goal that ends the game when a piece is captured.

```
(captured <piece-type>...<piece-type>)
```

Remarks

The **captured** goal is True when any one of players pieces of that piece-type is removed from the board. If multiple piece-types are listed, the goal will succeed when the first piece of any of the listed piece-types is captured.

A similar goal is pieces-remaining. It is identical to captured if there is only one of that piece type on the board and pieces cant be added to the board. If pieces can be added to the game, captured and pieces-remaining have a different effect. For example, if you want to determine whether a side is left with no Queens in a Chess-like game, you might try the goal (captured Queen). However, this does not take into account the fact that Pawn promotions may lead to a side having multiple Queens. In this case, a side may still have a Queen on the board even if one of its Queens is captured. To really measure whether all of a sides Queens are gone, pieces-remaining should be used here instead.

captured is also similar to checkmated, except that the piece must actually be captured to end the game.

Like stalemated, checkmated, and repetition, captured must be at the top goal level.

Usage

```
(game
...
  (loss-condition (White Black)
    (captured King) ; if your King is captured, you lose
  )
...
)
```

Examples

```
; True if your King (or one of your Kings) is taken
(captured King)

; True if one of your Kings, Queens, or Rooks is taken
(captured King Queen Rook)
```

See Also

checkmated pieces-remaining goal

cascade

Save part of a move to be generated.

cascade

Remarks

Use **cascade** to save out the move so far, before continuing the generate the rest of the move. The move will not be written out until an add is used. Using **cascade** lets you have multiple pieces move at once in a single game move. This makes it simple to program moves of multiple pieces, like castling in Chess.

cascade can be used in either a moves block or drops block. When used in a drops block, it allows multiple pieces to be dropped.

Right after a **cascade**, Zillions automatically sets "from position" for the next movement to the current position. If this is not what you want, you can use the from keyword to change it to something else. See below for examples.

It is possible to use more than one **cascade** in a move block before an add. In this way many pieces can be moved at once.

When a **cascade** is played, the movements of all the pieces occur simultaneously, so it is possible to have pieces move away from spots where other pieces land. If more than one piece are moved, the one moved in the first cascade is the key move, i.e. the one that the human player must pick up in order to execute the move. You should take care that two pieces aren't moved simultaneously to the same position, as the resulting behavior is undefined and likely to cause problems.

By default, after an add all the cascaded moves are remembered and applied to subsequent add's in that move block. This allows a move block to build up successively longer series of movements without having to regenerate them from scratch with every new move added. This behavior can be overridden by:

```
(option "discard cascades" true)
```

In any case, the list of cascaded moves will be cleared at the end of the move block.

Usage

```
(define O-O ; sample castling macro
   ( (verify never-moved?)
   e ; KB1
   (verify empty?)
```

```
   e ; KN1
   (verify empty?)
   cascade
   e ; KR1
   (verify (and friend? (piece? Rook) never-moved?) )
   from
   back ; K1
    ; Save expensive not-attacked?s for last
   (verify not-attacked?)
   e ; KB1
   (verify not-attacked?)
   to
   (set-attribute never-moved? false)
    ; We could check if KN1 is attacked too, but this isn't
    ; really necessary since Zillions doesn't allow any moves
    ; into check
   e ; KN1
   (set-attribute never-moved? false)
   add
   )
)
```

Examples

```
; Swap places with piece to the North
(n cascade (verify not-empty?) from s add)
```

See Also

from, to add option

change-owner, flip

Changes the owner of a piece.

```
flip
(flip <position>)
(flip <direction>)
change-owner
(change-owner <position>)
(change-owner <direction>)
```

Remarks

change-owner and **flip** are both used in a move definition to change which player a piece belongs to. Like capture, create and change-type, they affect the next move generated with an add. The position used is the current position, unless directed otherwise by an optional <position> or <direction> argument.

The change-owner keyword makes the piece so that its owned by the player who is moving, in effect "taking over" the piece. flip, on the other hand, cycles through the players, changing the owner to the "next" one in its cycle. In games with only two players, this is equivalent to swapping the owner, i.e. "flipping" the piece.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
     (n (flip n) (flip e) (flip w) add)
    ...
    )
  ...
  )
...
)
```

Examples

```
; Swaps the owner of a piece on the current position
flip

; Takes over a piece on the current position if it is owned by an opponent
change-owner
```

```
; Swaps the owner of a piece in the "n" direction
(flip n)

; Takes over a piece in the "n" direction if it is owned by an opponent
(change-owner n)

; Swaps the owner of a piece on the position e2
(flip e2)

; Takes over a piece on the position e2 if it is owned by an opponent
(change-owner e2)
```

See Also

capture create change-type

change-type

Changes a piece on the given position to a different piece type.

```
(change-type <piece-type>)
(change-type <piece-type> <position>)
(change-type <piece-type> <direction>)
```

Remarks

change-type is used in a move definition to promote (or demote) a piece to a different piece type. Like capture, create, flip and change-owner, it affects the next move generated with an add. The position used is the current position, unless directed otherwise by an optional <position> or <direction> argument.

Note that if you are changing the type of a piece the player is moving, there is a special syntax of add for this that you could use instead.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      ( n
        (change-type Knight n)
        (change-type Knight e)
        (change-type Knight w)
        add
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
(change-type Knight) ; Changes a piece on the current position to a Knight
(change-type Knight e2) ; Changes a piece on the position e2 to a Knight
(change-type Knight n) ; Changes a piece in the "n" direction to a Knight
```

See Also

capture create flip change-owner

checkmated

Stops the game when a side is checkmated.

(checkmated <piece-type>...<piece-type>)

Remarks

The **checkmated** goal is True when the one of players pieces of the given piece-type is threatened (check) and cant get out of the threat on his move. This models checkmating in standard Chess.

checkmated is really the same as captured except that in the real game moves played

1) you are not allowed to make a mistake about overlooking a direct threat (its illegal), and

2) the game is stopped early, before the piece is actually taken.

Like stalemated, captured, and repetition, **checkmated** must be at the top goal level.

Usage

```
(game
...
  (loss-condition (White Black)
    (checkmated King) ; if your King is checkmated, you lose (Chess)
  )
...
)
```

Examples

```
; True if your King (or one of your Kings) is checkmated
(checkmated King)

; True if one of your Kings, Queens, or Rooks is checkmated
(checkmated King Queen Rook)
```

See Also

stalemated goal

count-condition

Counts pieces to determine a winner when the game has ended.

```
(count-condition <players> stalemated)
(count-condition (total-piece-count <number>))
```

Remarks

count-condition tells Zillions to count total game pieces to determine the winner of a game. The side with the most pieces on the board wins. This is used in games like Reversi, whose goal is to have the most pieces at the end of the game.

You can use two arguments. Use stalemated to force Zillions to apply the count condition whenever a side has run out of moves. You also can specify a piece count limit, which forces Zillions to determine a winner once that number of pieces has been added to the board.

The counterparts to count-condition are win-condition, loss-condition, and draw-condition.

Usage

```
(game
...
 ; count pieces and determine winner once a player has no more moves
 (count-condition (White Black) stalemated)

 ; end this game once 16 piece have been placed on the board
 (count-condition (total-piece-count 16))
...
)
```

Examples

None.

See Also

stalemated win-condition, loss-condition, draw-condition

create

Creates a new piece on a position.

```
  version 2.0+ feature only
create
(create <position>)
(create <direction>)
(create <piece-type>)
(create <player>)
(create <piece-type> <position>)
(create <piece-type> <direction>
(create <player> <position>)
(create <player> <direction>)
(create <player> <piece-type>)
(create <player> <piece-type> <position>)
(create <player> <piece-type> <direction>)
```

Remarks

create is used in a move definition to add a piece to a position. If there is a piece already on the position, it will be replaced (captured). Like capture, change-type, flip and change-owner, it affects the next move generated with an add. <player> indicates the owner of the new piece. If left unspecified, it will default to the player who is moving. <piece-type> will default to the kind of piece whose moves are being generated. The position used is the current position, unless directed otherwise by an optional <position> or <direction> argument.

create is used as an additional side effect of a move or drop. If a move consists of nothing but adding a piece, you don't need to use **create**, as the add itself will place a piece when used with a drops block.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      ( (create trail)
        n
        add
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
create ; Duplicates the moving piece on the current square
(create Black) ; Places a copy of the moving piece, but owned by the Black playe
(create Knight w) ; Places a friendly Knight on the position to the w(est) direc
(create Black Knight e4) ; Puts a Black Knight on the e4 position
```

See Also

capture change-type flip change-owner

defended?

Determines whether the given position is defended.

```
defended?
(defended? <position>)
(defended? <direction>)
(defended? <move-type>)
(defended? <move-type> <position>)
(defended? <move-type> <direction>)
not-defended?
(not-defended? <position>)
(not-defended? <direction>)
(not-defended? <move-type>)
(not-defended? <move-type> <position>)
(not-defended? <move-type> <direction>)
```

Remarks

defended? returns true if the given position is defended by a friendly piece, otherwise false. In checkmate games, a piece defends a position even if moving to the position would put its side in check. To check attacks by an enemy piece use attacked? instead.

When a move-type is passed in only attacks of the given move-type are examined.

not-defended? returns false if the given position is defended by a friendly piece, otherwise true

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        ; Verify that the current position is defended by enemy
        (verify defended?)
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
(verify defended?) ; Verify that the current square is defended
(verify defended? n) ; Verify that position in direction n is defended
(verify not-defended? e2) ; Verify that position e2 is not defended

; Add a move only if the current square is defended by a friendly move
; of type jump-move
(if (defended? jump-move) add)
```

See Also

attacked?

default

Makes the game/variant the default variant for the ZRF.

```
(default)
```

Remarks

This means that when the ZRF is opened, this variant is initially loaded. There should only be only one default variant in the ZRF. If the default keyword appears in a game definition, it is not inherited by the variants. If there is no default variant explicitly specified, the first variant in the ZRF is the default.

Usage

```
(variant
...
  (default)
...
)
```

Examples

None.

See Also

variant game

description

Sets the description for the game variant.

```
(description <string>)
```

Remarks

When used in a the main body of a game, **description** defines what the user sees when chooses the Help:Description menu item. The default is no description, so if you want the user to know the object of the game, make sure you supply a description.

When used in a piece block, the **description** is shown in the Properties dialog for that piece. It should fully describe how that piece moves. If no help string is provided for the piece, then the description will also be displayed in the status bar instead of the help string.

Pass a string.

Usage

```
(game
...
  (description "The object is two get three thing-a-ma-bobs in a row.")
...
)

(variant
...
  (description "The object is to get four thing-a-ma-bobs in a row.")
...
)

(game
...
  (piece
  ...
    (description "thing-a-ma-bob\A thing-a-ma-bob can be placed on any red square
  ...
  )
)
```

Examples

```
(description
 "The object of this game is two get three thing-a-ma-bobs in a row
  before your opponent does. Captures occur only on the squares marked
  in purple with puke-green, zebra-stripped shading."
```

)

See Also

history strategy piece help

drops, moves

Defines how the piece moves.

```
(drops <drop-def>...<drop-def>)
(moves <move-def>...<move-def>)
```

Remarks

Zillions can generate moves in two basic ways: **drops** or **moves**. The **drops** keyword tells Zillions to loop through all board positions and apply the <move-def> rules to every position, regardless of whether a position is empty or not. The **moves** keyword tells Zillions to only apply these rules to all the pieces on the board that are of the current type and owned by the current side. **drops** is normally used for generating moves that drop a new piece on the board, while **moves** is used for moving pieces that are already on the board.

<move-def> may be either:

```
(move-type <move-type>)
(<instruction>... <instruction>)
```

<drop-def> includes both of those forms, plus these two additional forms:

```
(<position> <instruction>...<instruction>)
(<zone> <instruction>... <instruction>)
```

When a <drop-def> begins with a position, drops will only be looked at to that position, rather than to every position on the board. Similarly, when a <drop-def> begins with a zone, only drops to positions in that zone will be looked at. These forms are quicker than letting Zillions loop through all the positions and discarding irrelevant ones yourself.

In a **moves** statement, a generated add instruction will generally result in the moving of a friendly piece on the board. In a **drops** statement, an add will move one piece from the off-board store to the destination position. You can specify how many pieces are initially off the board using the off keyword in a board-setup statement. You can further control how the pieces off the board are maintained using the "recycle captures" or "recycle promotions" option.

A piece may have both a **moves** and **drops** keyword, but it makes no sense for it to have more than one of either.

Usage

```
(game
...
  (piece
```

```
    (drops
    ...
    )
    (moves
    ...
    )
  )
)
```

Examples

None.

See Also

<instruction>

dummy

Identify a dummy piece or dummy position..

(dummy)
(dummy <position>...<position>) ***version 1.0.2+ feature only***

Remarks

Occasionally you may want to use a piece type or position as a place-holder in your game, not as a real piece or position in the game. The **dummy** keyword tells Zillions that the piece or position is a dummy piece or position and the player of the game should not interact with it.

Move actions involving dummy pieces are not displayed on the move list. Dummy pieces are not shown in the pop-up menu for editing the board. A dummy piece normally sits on a dummy position and doesn't have an associated image; however if it does have an image and rests on a non-dummy position, it will be displayed. A dummy piece can have moves, but if it does, it is recommended that it be owned by a random player, because allowing a human player to move dummy pieces defeats the purpose.

Move actions involving dummy positions are not displayed on the move list. Dummy positions never display a notation label and any piece on a dummy position is not drawn. When a force-feedback mouse is used, dummy positions are not felt as part of a selection screen. Dummy positions are treated as valid move destinations and can be linked to from other positions. on-board? will still return True when referencing a dummy position. Dummy positions can be used as "holders" for pieces. They are not meant to be used as part of the displayable board that people play with.

dummy should be used without arguments in a piece definition or with arguments in a board definition. When used with positions, these positions may either have been defined elsewhere or not. If not, then the **dummy** statement will create new position(s) with a null location.

Usage

```
(game
...
  (board
  ...
    ; These positions are not on the playable board
    (dummy invisible-1 invisible-2)
  ...
  )
  (piece
  ...
    (dummy) ; This is not a real game piece
  ...
  )
```

...
)

Examples

None.

See Also

piece board

engine

Indicates that a plug-in DLL engine should be used in place of the generic Zillions engine for this variant.

```
(engine <file-string>)
```

Remarks

Zillions of Games contains a universal gaming engine (or "AI") capable of playing any game that can be specified by a ZRF. However, Zillions also supports a plug-in architecture that allows external engines, written for specific games or variants, to be used instead of the Zillions engine. These plug-ins take the form of DLLs adhering to an API described here. Using **engine** in a ZRF means that the referenced DLL should be used instead of Zillions normal engine.

Note that plug-in engines assume the rules of particular variants and do not have direct access to the ZRF itself. Therefore you should be extremely cautious modifying ZRFs that use external engines. A plug-in engine attempting to play a game it no longer understands the rules for may crash. If Zillions determines that a plug-in engine has searched and come up with an illegal move, Zillions' universal engine will take over play until the player leaves the variant.

Usage

```
(game
...
  (engine "thinker.dll") ; Searches using an engine called "thinker"
...
)
```

Examples

None.

See Also

None.

from, to

Sets the move's "from" or "to" position.

from
to

Remarks

Normally the "from" position is where the piece you are generating moves for is. The "to" position is the position you are at when you do an add. Using the **from** and **to** keywords in the move language allows you to change this.

from makes the current position the "from" position. This allows weird effects, such as a piece that can allow other pieces to move in different ways. You should take care that the "from" position is not empty, as adding such a move is not allowed.

to makes the move language more concise and move processing more efficient. For example, you might know the move is to the current square, but you may need to go elsewhere first to check the move's validity.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      ( ... from ... )
    ...
    )
  ...
  )
...
)
```

Examples

```
; Moves the piece west of the current square 2 squares to the East
w (verify friend?) from e e add
```

See Also

add

game, variant

Defines a game variant.

```
(game <game-def>...<game-def>)
(variant <game-def>...<game-def>)
```

Remarks

A **game** construct defines the rules of a game, and consists of the parts shown below. It is at the highest level of a .zrf file. There may be only one **game** in each rules file.

A **variant** construct defines a variation on the game defined in the rules file. Any <game-def> included in a **variant** replaces or "overrides" the equivalent section defined in the game. Note that when a **variant** has one or more end-conditions (win-condition/loss-condition/draw-condition) then Zillions will use only those conditions (ignoring the end-conditions in the main **game** section). If the **variant** doesn't have any end-conditions then Zillions will use the game's end-conditions.

Any **game** or **variant** construct must have a title within it, since this determines the name of the menu item. Any variant with the title of a dash:

```
(variant (title "-"))
```

will show up on the Variant menu as a separator line. This is good for organizing lots of variants of different types, such as variants on different boards.

A <game-def> can contain any one of the following sections:

```
(board ... )           ; defines the playing board
(board-setup ... )     ; defines the positions of pieces at the start of the game
(capture-sound ... )   ; which sound to using during animated captures
(change-sound ... )    ; sound used when animating piece flips and owner changes
(click-sound ... )     ; sound played when user clicks on a pieces
(count-condition ... ); defines a game-over condition where the pieces are counte
(default)              ; the variant that is initially loaded
(description ... )     ; what text to display when the user selects Help, Descript
(draw-condition ... ) ; defines a condition for a drawn (tied) game
(draw-sound ... )      ; the sound made when the game is a draw
(drop-sound ... )      ; the sound made when animating a drop of a piece onto the
(loss-condition ... ) ; defines a condition for losing the game
(loss-sound ... )      ; sound played when the user loses
(move-priorities ... ); defines a priority order that must be followed for selec
(move-sound ... )      ; the sound of an animated move  of a piece
(music ... )           ; which MIDI file to play for background music
(opening-sound ... )   ; sound played when a variant is loaded or when a new game
(option ... )          ; sets a predefined option
(piece ... )           ; defines a piece to use in the variant
```

```
(players ... )          ; lists the players (sides) which will play in this variant
(release-sound ... )    ; sound played when the user finishes dragging a piece
(title ... )            ; the title of this variant
(turn-order ... )       ; the order (turns) in this variant
(win-condition ... )    ; defines a condition for winning the game
(win-sound ... )        ; sound played when the user wins
```

Usage

```
(game
...
)

(variant
...
)

(variant
...
)
```

Examples

None.

See Also

None.

go

Changes the current position.

```
(go from)
(go to)
(go mark)
(go last-from)  version 1.2+ feature only
(go last-to)  version 1.2+ feature only
```

Remarks

go changes the current position. Use **go** in a move description to return to a previous from, to or marked position while generating moves. Being able to jump to the from and to squares will often free up the mark for other uses.

You can also use **go** to jump to the "from" or "to" squares of the previous move using last-from and last-to. Note that the generation for the move block will immediately cease if such a position doesnt exist, just as if you had tried to follow a non-existent directional link. See last-from? or last-to? for information about how to test a position to see if its the last-from or last-to position.

Usage

```
(game
...
  (piece
  (name "Queen")
  (help "Queen: can drop anywhere another Queen cannot attack.")
  (drops (
    (while (empty? n) n)
    (verify (not-friend? n))
    (go from)
  ...
  ))
...
)
```

Examples

```
(go from) ; Goes to the "from" square
(go to) ; Goes to the "to" square
(go mark) ; Goes to the marked square  identical to "back"
(go last-from) ; Goes to the "last-from" square
(go last-to) ; Goes to the "last-to" square
```

See Also

last-from? last-to? from to mark back

grid, start-rectangle, dimensions, directions

Define a linked set of positions.

```
(grid <grid-arg>...<grid-arg>)
(start-rectangle <left> <top> <right> <bottom>)
(dimensions (<string> (<x> <y>)...(<string> (<x> <y>))
(directions <directions-arg>...<directions-arg>)
```

Remarks

Use **grid** to quickly construct a pattern of linked positions within a board block. This is a lot easier than using a number of positions and links commands. Within a **grid** block, you specify these three parts:

start-rectangle defines the left, top, right, and bottom pixels of a single position on the board, typically the upper-left position. From this starting rectangle, the bounding rectangles of other positions are derived.

dimensions indirectly defines how many positions the grid statement should create, how these positions will be named, and the x and y pixel offsets for the new positions. **dimensions** should contain one block (argument enclosed in parentheses) for every dimension on the board. So, for a two-dimensional board there will be two blocks. Each block has a string, which defines the notation for each step in that dimension, separated by "/" delimiters. The notation for a position, which is displayed to the user, is the concatenation of the notations for each dimension. Next comes an (<x>, <y>) pair, which determines the x and y pixel offsets on the screen for every step in that direction.

directions defines how the positions are linked and names the directions. It may contain any number of <direction-arg> blocks, which have the form:

```
(<direction-name> <offset-dimension-1>...<offset-dimension-n>)
```

<direction-name> gives a name to the direction that you can refer to elsewhere in the ZRF. Next comes a list of offsets, one per dimension, which define how many steps to take in that dimension. Note that these are the same step increments used in the **dimensions** statement, not pixel increments.

A game may include multiple grids. Each grid is linked up only with the directions it defines. If you want more than one grid to be linked with the same direction, then repeat the definition of the direction in each grid. It is possible to define the same direction differently (with different offsets) in two grids. In this case, only one direction is created, but it will link positions differently in each grid.

Usage

```
(game
```

```
...
(board
  ...
  ; 3-dimensional board
  (grid
   start-rectangle 4 4 31 31)
  (dimensions ; 4x4x4
    ("I-/II-/III-/IV-" (0 80)) ; offset down
   ""a/b/c/d" (58 0)) ; offset right
   "1/2/3/4" (29 14)) ; offset right and down
  )
   directions
    (n -1 0 0) (e 0 1 0) (nw -1 -1 0) (ne -1 1 0)
    (up 0 0 1) (upn -1 0 1) (upe 0 1 1)
    (ups 1 0 1) (upw 0 -1 1) (upne -1 1 1)
    (upnw -1 -1 1) (upse 1 1 1) (upsw 1 -1 1)
  )
  )
...
)
```

Examples

```
; Definition of a standard 8x8 checkerboard
(grid
  ; The pixel dimensions of the top-left square
  (start-rectangle 4 4 37 37)

  ; We want a 2-dimensional board.
  ; The notation we want is:
  ; a8 b8 c8 d8 e8 f8 g8 h8
  ; a7 b7 c7 d7 e7 f7 g7 h7
  ; a6 b6 c6 d6 e6 f6 g6 h6
  ; a5 b5 c5 d5 e5 f5 g5 h5
  ; a4 b4 c4 d4 e4 f4 g4 h4
  ; a3 b3 c3 d3 e3 f3 g3 h3
  ; a2 b2 c2 d2 e2 f2 g2 h2
  ; a1 b1 c1 d1 e1 f1 g1 h1
  ; Our starting rectangle was for a starting square in
  ; the top-left, which corresponds to a8. Therefore
  ; a8 will come first in our notation list. We want our
  ; files (columns) and ranks (rows) to be 32 pixels apart.
  (dimensions
    ("a/b/c/d/e/f/g/h" (32 0)) ; files
    ("8/7/6/5/4/3/2/1" (0 32)) ; ranks
  )

  ; Link positions on the diagonals, which is all we値l need
  ; for Checkers. To move southeast we move one-square away
  ; from the starting square (a8) in both dimensions, i.e.    1 1
  (directions (ne 1 -1) (nw -1 -1) (se 1 1) (sw -1 1))
)
```

See Also

board positions links

help

Associate status bar help with a piece type.

(help <string>)

Remarks

Use **help** to specify the text to be displayed in the Status bar when the cursor moves over a piece. This should be information about how that piece moves, to make it easier for people to learn new games.

Pass a string. As the string is displayed only on one line, backslashes have a different formatting meaning than in other strings. Text before a backslash ("\") is left-justified in the status bar, text after a single backslash is centered, and text after 2 backslashes is right-justified.

Usage

```
(game
...
  (piece
    (name "Rook)
    (help "Rook: slides any number of squares along the row or column.")
  ...
  )
...
)
```

Examples

```
(help "Flibber: Moves totally at random")
(help "\This is an unmovable wall.")
```

See Also

piece

history

Sets the history for the game variant.

```
(history <text>))
```

Remarks

history defines what the user sees when he asks for a history of the current game under Help, History. Pass a string.

The default is no description, so if you want the user to know the history of the game, make sure you use **history**.

Usage

```
(game
...
  (history "This game was invented by John Smith.")
...
)

(variant
...
  (history "This game was invented by Fred, John Smiths brother.")
...
)
```

Examples

```
(history
  "Played on the decks of Viking ships in the days of Erik the Red."
)
```

See Also

description strategy

if, else

A conditional expression.

```
(if <condition> <instruction>...<instruction>)
(if <condition> <instruction>...<instruction> else <instruction>...<instruction>)
```

Remarks

The **if** construct in the move language can be used to execute move language instructions conditionally. If the condition evaluates to True then the move instructions after the condition are executed. If the condition evaluates to False then the instructions after the else (if any) are executed.

if is very useful in testing certain things out without halting the present move generating (which verify does if it fails its test).

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        (if empty? n add)
      ...
      )
    )
  ...
  )
...
)
```

Examples

```
; If the position is empty go in the "n" direction and add the move
(if empty? n add)

; If the position is empty go in the "n" direction and add the move,
; otherwise set the flag "my-flag" to True
(if empty? n add else (set-flag my-flag true))
```

See Also

<condition> <instruction>

image

Sets an graphic to use.

```
(image <file-string>...<file-string>)
(image <image-def>...<image-def>)
```

Remarks

When used in a board construct...

```
 (image <file-string><file-string>)
```

When used in a piece construct...

```
 (image <image-def>...<image-def>)
```

The **image** construct is used to associate bitmap graphics (*.BMP) with a board or piece. A <file-string> is the bitmap file to use and can contain a path name.

<image-def> relates a player to an image for that piece type. It takes the form:

```
 <player> <file-string><file-string>
```

This allows you to use different colored pieces for different sides.

By supplying more than one <file-string> for a board or players piece, the game designer can give the user a choice of graphics. When the user chooses "Switch Piece Set", Zillions cycles through the bitmaps. It is not necessary to have the same number of bitmaps for the board and each players piece type. For example, you might have a single set of piece bitmaps, but several board bitmaps available. In this case, Zillions would display a new board every time the user chooses to switch piece sets.

For pieces, use solid green (value of 255) for any areas you wish to be "transparent" when the piece is drawn.

Usage

```
(game
...
  (board
  ...
    (image "myfile.bmp")
  )
...
)

(game
```

```
...
  (piece
  ...
    (image White "WhitePawn.bmp" Black "BlackPawn.bmp")
  )
...
)
```

Examples

None.

See Also

board piece

kill-positions

Deletes positions from the board.

```
(kill-positions <position>...<position>)
```

Remarks

kill-positions goes within the board construct and is processed after all other board keywords. The position is completely unlinked from all other positions and then destroyed. **kill-positions** is generally used to delete unwanted positions created with a grid statement. Deleting a position frees up its memory and can speed up move generation.

Usage

```
(game
...
  (board
  ...
    (kill-positions c3 f6 d7)
  )
...
)
```

Examples

```
(kill-positions c3) ; Deletes position "c3"
(kill-positions c3 f6 d7) ; Deletes positions "c3", "f6", and "d7"
```

See Also

unlink board

last-from?, last-to?

Compares the position to the destination of the last move or drop.

```
last-from?
(last-from? <position> <position>)
(last-from? <direction> <direction>)
last-to?
(last-to? <position>)
(last-to? <direction>)
not-last-from?
(not-last-from? <position>)
(not-last-from? <direction>)
not-last-to?
(not-last-to? <position>)
(not-last-to? <direction>)
```

Remarks

last-to? compares the position to the destination of the last move or drop. **last-from?** compares the position to the starting position of the last move. If the move is something other than a move or drop, the value will be false. Can be used within verify, if, etc. in the move language for special move generation like en-passant captures.

To jump directly to the last-from or last-to position, if it exists, use go.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (n (verify last-to?)...
    ...
    )
  ...
  )
...
)
```

Examples

```
last-to? ; Is current position same as the last destination?
(last-to? a3) ; Was the last destination a3?
(last-to? ne) ; Was the position ne of here the last destination?
```

```
(last-from? a3) ; Was the last move a move from a3?
```

See Also

go from, to

links

Creates one-way links in a direction between pairs of positions

```
(links <direction> (<position> <position>) ... (<position> <position>))
```

Remarks

The **links** keyword is used to define directions. It goes at the top level in a board definition. It's okay to have multiple **links** statements with the same direction. If you need to be able to move a piece from one location and back, make sure you link it in both directions.

Note that a short-hand for defining links is the grid construct, which can define a whole set of positions and links at once for boards with uniformly-spaced positions.

Usage

```
(game
...
  (board
  ...
    (links n (a1 a2) (a2 a3))
  ...
  )
...
)
```

Examples

```
; Declares a direction "n" and creates an "n" link from
  a1 to a2
(links n (a1 a2))
 ; Note this does not add any links from a2 back to a1

; Declares a direction "n" and creates an "n" link from a1 to a2,
; and from a2 to a3
(links n (a1 a2) (a2 a3))
```

See Also

grid board

move-priorities

Assign priorities to move-types.

```
(move-priorities <move-type>...<move-type>)
```

Remarks

Use **move-priorities** to force Zillions to only allow certain types of moves to be generated. The <move-type> arguments should be names defined in a pieces move-type statements. The first move type in the list will have preference over the remaining types. Thus, if a move of the first type exists it must be played. If no moves of the first type exist, then a move of the second type must be played, and so on. Moves without an explicit move-type have the lowest priority.

Usage

```
(game
...
  (move-priorities move-type1 move-type2 move-type3)
)
```

Examples

Suppose your game requires that a side must capture if there are any captures available to be made. You might define your piece with these moves and move types:

```
  (moves
    (move-type capturing)
    (n (if enemy? add))
    (move-type non-capturing)
    (n (if empty? add))
  )
```

And specify that capturing type moves must be played before non-capturing moves:

```
  (move-priorities capturing non-capturing)
```

See Also

move-type

move-type

Classifies sets of moves.

```
(move-type <move-type>)
```

Remarks

move-type is used to classify sets of moves, for instance, capturing and non-capturing moves. Use within a moves or drops construct. Every move of the piece defined after the **move-type** statement will be of that type. The <move-type> argument should be a unique name you invent to represent that move type.

Move types are useful in four different ways:

- move-priorities uses them to give priority to some kinds of moves over others
- turn-order can use them to allow only certain kinds of moves to be made
- add-partial can use them to restrict continuation moves to a certain type
- attacked? and defended? can use them to look for only certain kinds of attacks

Usage

```
(game
...
  (piece
    ...
    (moves
    ...
      (move-type straight-moves)
      ...
      (move-type twisty-moves)
    ...
  )
...
)
```

Examples

None.

See Also

move-priorities turn-order add-partial attacked? defended?

music

Specify a MIDI file to play in the background when "Music" is turned on in the "Options..." window.

```
(music <file-string>)
```

Remarks

The music will automatically start playing when the variant is loaded.

Usage

```
(game
...
  (music "mymusic.mid")
...
)
```

Examples

None.

See Also

-sound

name

Names a piece type.

```
(name <name>)
```

Remarks

name specifies defines a symbol, what the piece type or zone is called in the move language. A piece-type **name** may also be displayed to the user, though it is different from the notation, which is a string only used for display purposes.

Usage

```
(game
...
  (piece
  ...
    (name King)
  ...
  )
...
)
```

```
(game
...
  (board
  ...
    (zone
      (name promotion-zone)
    ...
    )
  ...
  )
...
)
```

Examples

None.

See Also

notation piece zone

not

Negates a condition or goal.

```
(not <condition>)
(not <goal>)
(not <piece-type>)
(not (opponent <piece-type>))
(not (any-owner <piece-type>))
```

Remarks

The **not** modifier may be used in three separate circumstances:

1. As part of a <condition>, such as:

```
(set-flag my-flag (not (piece? King)))
```

2. To negate some goals, such as:

```
(not (pieces-remaining 1))
```

3. To define occupants inside an absolute-config goal, such as:

```
(absolute-config (not rook) (a1 h1))
```

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
       (verify (not (piece? King)))
    ...
    )
  )
  (win-condition (Black)
    (not (pieces-remaining 1))
  )
  (win-condition (White)
    (absolute-config (not Rook) (a1 h1))
  )
)
```

Examples

None.

See Also

<condition> <goal> absolute-config

notation

Specifies a compact piece name for this piece type.

```
(notation <string>)
```

Remarks

notation is used for display purposes. If a piece notation is needed, but none is defined for the piece-type, the name will be used instead.

Usage

```
(game
...
  (piece
  ...
    (notation "K") ; Short for King
  ...
  )
...
)
```

Examples

None.

See Also

name

open

Specifies a variant to load when a piece is clicked.

```
(open <file-name>)
```

Remarks

The <file-name> argument specifies either:

- a ZRF file to open when the piece is clicked. If you append a greater-than sign (">") plus the name of a variant, that variant will be loaded. Otherwise the default variant -- the one defined by a "game" tag -- will be loaded. Using **open** in this way allows you to create game selection screens, making it easier to find an appropriate game. Zillions considers any game containing the definition of a piece that opens a ZRF a selection screen, unless the game specifically overrides this with the "selection screen" option.

- a URL to open when the piece is clicked. The computer's default web browser, mail program, or ftp program is invoked with the specified address. To be recognized as a URL, the string must start with either "http://", "https://", "mailto:", or "ftp://".

Positions containing pieces with an **open** are ignored when right-click editing. When rotating through pieces using tab-right-click, pieces with an **open** are skipped.

Usage

```
(game
...
  (piece
  ...
    (open "Beginner.zrf")
  ...
  )
)
```

Examples

```
(piece
  (name pBeginner)
  (notation "Beginner Games")
  (description "These games have easier to learn rules.")
  (image You "images\Select\beginbut.bmp")
  (open "Beginner.zrf") ; Open Beginner.zrf
)
```

```
(board-setup
  (You (pBeginner a1))
  ...
)

; Open the Extinction Chess variant in Chess.zrf
(open "Chess.zrf>Extinction Chess")

; Contact the author (with praise for the game?)
(open "mailto:theauthorguy@some-random-domain-name.com")

; Open up the game's home page
(open "http://www.some-random-domain-name.com/mygreatgame.html")
```

See Also

piece option

opposite

Steps in the opposite direction.

(opposite <direction>)

Remarks

Use **opposite** to step in the opposite direction to the argument. For example, if "n" is north, then (**opposite** n) would step south on a standard chess board. You can use this command to return from the direction you just came. You may not need to define as many directions if you use **opposite**.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        (opposite n)
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
(define move-2 (if (empty? $1) $1 add (opposite $1)) )
(game
...
  (piece
    (moves
      (n (move-2 w) (move-2 e) )
    )
  ...
  )
...
)
```

See Also

<instruction>

option

Set an option for the variant.

```
(option <option-string> <option-value>)  version 2.0+ feature only
```

Remarks

option allows you to set up certain features in your games. These options can override user interface settings or the way Zillions generates moves. For <option-string> pass in the string denoting a known option, such as "animate captures". For <option-value> pass in the new value for the option, usually **true** or **false**.

The allowable <option-string> arguments are described below, along with their possible <option-value> settings. The default <option-value> settings are shown in red. "animate captures", "animate drops", "show moves list", and "smart moves" default to the value **default**, which means their behavior is determined by user settings. Other features default to **false** (off).

"animate captures"
"animate drops"

Overrides the "Animate Pieces" user setting in the Options dialog.
true: Captures and drops will always be animated.
false: Captures and drops will never be animated.
default: Captures and drops will be animated if and only if "Animate Pieces" is checked in the user options.

"discard cascades"

Determines whether the list of accumulated cascade's will be reset after each add in a move block or will continue to accumulate. See When Move Data Is Reset.
true: Cascades will be discarded when an add is complete. A second add following the first will not have any of the cascades encountered before the first add.
false: Cascades will be retained after an add to be used in subsequent add's later in the move block.

"highlight goals"

In authoring mode, overrides the option to show goals, which places stars on positions that appear in absolute-config goals, i.e. those positions where goal-position? returns true. The "Goals" option will appear in the "Show" right-click menu only when the "Use authoring Show options" authoring setting is checked in the Options dialog. This option also allows goals to be displayed outside of authoring mode.

true: Goals are highlighted in this variant, except when the F9 key is down.
false: Goals are not highlighted in this variant, unless the F9 key is down and you are in authoring mode..
default: If the Show:Goals menu item is available (you are in authoring mode and "Use authoring Show options" is checked), then goals are highlighted in this variant according to the Show:Goals setting. Otherwise, goals are not highlighted. Holding down F9 key toggles the goal highlighting in either case.

"include off-pieces"

Changes whether pieces off the board count in a pieces-remaining goal.
true: Include the count of pieces off the board and not yet dropped when evaluating the pieces-remaining goal.
false: Ignore the off-board pieces when evaluating the pieces-remaining goal.

"maximal captures"

Changes whether players are forced to make moves with the most partial move continuations.
true: Force players to make moves with the most partial move continuations. For example, this can be used to implement the rules for draughts games where players must play capturing moves that take most pieces.
false: Players may make any partial move.
2: Same as "true" above except that if there are two or more continuations that are equally long, Zillions only allows the continuation that takes the higher-valued types of pieces. The value of pieces is determined internally and does not change during the game.

"pass partial"

Changes whether a player is allowed to end his move even if he has subsequent partial moves available.
true: A player may pass, ending his move without completing all partial move continuations .
false: A player must complete al partial moves.

"pass turn"

Changes whether a player is allowed to pass his turn.
true: A player may pass.
false: A player may not pass.
forced: A player may pass if and only if he has no moves, e.g. Reversi, Blobs.

"prevent flipping"

Controls whether the user is permitted to flip the board display. This allows you to disable flipping of the board in games where it doesn't make sense, such as "Towers of Hanoi", or where the board is better viewed without inversion.
true: The "Flip Board" menu item is disabled. Nothing may be flipped.

false: The "Flip Board" menu item is enabled as normal. It flips the positions and inverts the board bitmap.
2: The "Flip Board" menu item is enabled, but selecting it flips the positions without inverting the board bitmap.
3: Like 2, except a different method is used for calculating the flipped positions. Firstly, dummy positions and positions that start out with an openable piece (a piece with an open string) are ignored. Secondly, whereas normally positions are flipped in relation to the dimensions of the board bitmap, here positions are flipped in relation to the minimum bounding area of all (non-ignored) positions. What this means is that borders outside the actual playing area will be ignored. This option is useful if you have asymmetrical borders around a symmetrical board or if you have an openable piece that shouldn't be moved when the board is flipped.

"progressive levels"

Allows you to specify that the player progresses through variants in a sequence.
true: As soon as the player wins, the next variant in the ZRF is loaded.
false: Winning does not change the variant.

"recycle captures"

Changes whether captured pieces can be re-dropped.
true: Any piece that is captured is returned to the off-board store, where it can be dropped on the board again.
false: Captured pieces disappear and are not recycled.
Note that you can specify how many pieces are initially off the board using the off keyword in a board-setup statement.

"recycle promotions"

Changes whether promoted/changed pieces come from an off-board store.
true: Pieces are taken from the off-board store when they are used in a promotion (see add). For example, if a Pawn promotes to a Queen in a chess variant, the Queen is taken from the store of Queens. If there are no Queens in the store, the Pawn may not make that promotion. This feature also affects change-type actions as well: when a piece is changed into a different type of piece using change-type, the new piece is taken from the off-board store.
false: Pieces may be promoted without regard to the off-board store.
Note that you can specify how many pieces are initially off the board using the off keyword in a board-setup statement.

"selection screen"

Overrides whether the variant is considered a selection screen. When a selection screen is open Zillions doesn't generate moves, check goals, or show position labels. The moves list, toolbar, and search bar are hidden and their menu items are disabled. The chat bar is also hidden. The Save Game As, Print Moves List, Print Preview, and Give Hint menu items are disabled. Tooltips are shown for pieces. The positions will be felt if the "selections screens" option is

checked in the Feel options tab. The "to move" text of the status bar is set to "No game open" and right-clicking off a position will say "Click on a picture to open a game".
true: The game is treated as a selection screen.
false: The game is not treated as a selection screen.
default: If a piece exists with an open block, the game is considered a selection screen; otherwise it won't be..

"show moves list"

Overrides the "Show Moves List" user setting in the View menu.
true: The moves list is shown (and the "Show Moves List" menu item is checked) when the variant is loaded.
false: The moves list is hidden (and the "Show Moves List" menu item is unchecked) when the variant is loaded.
default: The moves list will be shown or hidden according to the user's preference and whether the variant is a Select Screen.

"silent ? moves"

Specifies whether random players (those whose names start with "?") make moves silently.
true: The moves of random players never produce sounds.
false: The moves of random players may or may not produce sounds based on the "Sounds" setting in the Options dialog..

"smart moves"

Overrides the "Smart Moves" user setting in the Options dialog.
true: "Smart Moves" are always enabled for this variant.
false: "Smart Moves" are always disabled for this variant.
default: "Smart Moves" are enabled or disabled based on the "Smart Moves" setting in the Options dialog.

Usage

```
(game
...
  (option "smart moves" true)
...
)
```

Examples

```
(option "prevent flipping" true)     ; It's a puzzle; there's no point in flippin
(option "progressive levels" true)   ; Once he's finished, put him on the next le
(option "animate captures" default) ; Was set to true in the game, but in this
```

```
;  variant we want to use its original value
```

See Also

add board-setup pieces-remaining

piece

Describe a piece type.

(piece <piece-arg>...<piece-arg>)

Remarks

piece defines what a piece looks like (image), how it can be referred to within the ZRF (name), the name that the player sees (notation), how it can move (moves, drops), and any attached game-specific information (attribute).

<piece-arg> may be:

```
(attribute true/false)
(description <string>)
(drops <move-def>...<move-def>)
(dummy)
(help <string>)
(image <image-def>...<image-def>)
(moves <move-def>...<move-def>)
(name <string>)
(notation <string>)
(open <string>)
```

Please follow these links for a further description of the arguments.

Usage

```
(game
...
  (piece
    (name Wall)
    (image Neutral-Player "Wall.bmp")
  )
...
)
```

Examples

```
(define max-slide
  ($1 (verify empty?) (while (empty? $1) $1) add)
)
(piece
  (name Neutron)
  (help "Neutron: slides in a straight line until it hits edge or piece")
```

```
  (description "Neutron\Slides in a straight line in any direction like a
    Queen in Chess. The Neutron must move as far as it can; it continues
    until its blocked by a wall or another piece. There is no capturing.")
  (image Neutral "images\Neutron\Neutron.bmp")
  (moves
    (max-slide n) (max-slide e) (max-slide s) (max-slide w)
    (max-slide ne) max-slide nw) (max-slide se) (max-slide sw)
  )
)
```

See Also

None.

pieces-remaining

Stops the game when a player has a specific number of pieces.

```
(pieces-remaining <number>)
(pieces-remaining <number> <piece-type>)
```

Remarks

The **pieces-remaining** goal is true when the players pieces on the board becomes equal to the given number. Optionally, you may include a second argument specifying the type of piece. To total everybody's pieces use total-piece-count instead.

If you want to also count the pieces in the off-board store, use the "include off-pieces" option.

Usage

```
(game
...
  (win-condition (White)
    (pieces-remaining 0 Knight)
  )
...
)
```

Examples

```
; True when the side is reduced to 3 pieces on the board
(pieces-remaining 3)

; True when the side has no Knights on the board
(pieces-remaining 0 Knight)
```

See Also

total-piece-count <goal> option

players

Defines the names of the players.

(players <player>...<player>)

Remarks

When used in a game or variant block, **players** defines the names of the players in that variant. The defined player names can then be used elsewhere in the ZRF. Neutral players, players that dont have a turn, should be listed here along with other players. To create a random player, a hidden player who moves randomly, begin its name with a question mark.

players is also used in a zone block to specify which sides the zone applies to.

Usage

```
(game
...
  (players White Black) ; White and Black are playing
...
)

(game
...
  (board
  ...
    (zone
      (players White)
    ...
    )
  ...
  )
...
)
```

Examples

None.

See Also

game, variant zone

position-flag?

Tests a position-flag.

```
(position-flag? <flag-name>)
(position-flag? <flag-name> <position>)
(position-flag? <flag-name> <direction>)
(not-position-flag? <flag-name>)
(not-position-flag? <flag-name> <position>)
(not-position-flag? <flag-name> <direction>)
```

Remarks

Returns the value of the given flag for the given position. A **position-flag** may be set using set-position-flag.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        ; Verify that my-flag is true
        (verify (position-flag? my-flag))
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
; An example usage from Ultima
; A Coordinator is a piece that captures any piece that is on
; the same rank or file as the Coordinators destination AND
; on the same rank or file as the friendly King. In other
; words, the Coordinator forms a rectangle on the board with
; the King and pieces at the other two corners of the rectangle
; are captured.
; This macro sets the position flag king-line on every square in
```

```
; a straight line in direction $1
(define make-king-line
  (while (on-board? $1) $1 (set-position-flag king-line true))
)

...
; This macro extends in a straight line, capturing every enemy
; piece it encounters if the square also has the king-line
; position flag set
(define coordinate
  (while (on-board? $1) $1
    (if (and enemy? (position-flag? king-line)) capture)
  )
  (go to)
)

...
; At the beginning of a Coordinators move, the king-line position
; flag is set for all squares on the same rank or file as the King
  ; Start on King square and then...
 (make-king-line n) back
 (make-king-line e) back
 (make-king-line s) back
 (make-king-line w)

...
; Before a Coordinators move is added, captures are looked for on
; all orthogonal directions.
 (if (not-position-flag? king-line)
   (coordinate n) (coordinate e) (coordinate s) (coordinate w)
 )
 add
```

See Also

set-position-flag

positions

Defines new positions on the board.

```
board:
(positions
 (<position> <left> <top> <right> <bottom>)
 ...
 (<position> <left> <top> <right> <bottom>)
)

zone:
(positions <position>...<position>)
```

Remarks

positions can be put in a board definition to create new positions. The left/top/right/bottom arguments define the rectangle of the position in pixels on the board. Zillions uses this rectangle to place pieces and allow uses to select positions with the mouse. Note that a short-hand for defining links is the grid construct, which can define a whole set of positions and links at once for boards with uniformly-spaced positions.

positions is also used in a zone construct to define the set of positions that is the zone.

Usage

```
(game
...
  (board
  ...
    (positions
      (a1 16 16 112 112)
      (a2 16 128 112 224)
    )
  ...
  )
...
)

(game
...
  (board
  ...
    (zone
      (positions a8 b8 c8 d8 e8 f8 g8 h8)
    ...
    )
  ...
  )
```

```
...
)
```

Examples

```
; Defines two new positions called "a1" and "a2" and their corresponding
; pixel rectangles on the screen.
(positions (a1 16 16 112 112) (a2 16 128 112 224))
```

See Also

board grid zone

relative-config

Stops the game when pieces are in a certain configuration.

(relative-config <check-occupant>...<check-occupant>)

Remarks

The **relative-config** goal evaluates to True if pieces are in certain configurations relative to each other. If the configurations are restricted to specific positions on the board, use absolute-config instead.

<check-occupant> may be one of the following:

<direction>	Move in a direction
<piece-type>	Do I have a piece of this type here?
(opponent <piece-type>)	Does someone else have a piece of this type here?
(any-owner <piece-type>)	Does anybody have a piece of this type here?

Some of these may be negated with a "not" as follows:

```
(not <piece-type>)
(not (opponent <piece-type>))
(not (any-owner <piece-type>))
```

Usage

```
(game
...
  (win-condition (White Black)
    (relative-config X-piece n X-piece n X-piece)
  )
...
)
```

Examples

```
; True if the side has a Counter on top of another Counter
(relative-config Counter n Counter)

; True if the side has 3 Counter in a row vertically
(relative-config Counter n Counter n Counter)

; True if the side has a Counter 2 positions above another one of his Counters.
```

```
; Note that anything may be in the middle position
(relative-config Counter n n Counter)

; True if the side has 4 Balls that form a square
(relative-config Ball n Ball e Ball s Ball)

; True if 2 of sides Pawns sandwich an opponents Knight on a horizontal line
(relative-config Pawn e (opponent Knight) e Pawn)

; Checks for the absence of a friendly Pawn, then "n" of that for an enemy Pawn,
; then "n" of that for the absence of an enemy Pawn.
(relative-config (not Pawn) n (opponent Pawn) n (not (opponent Pawn)))
```

See Also

absolute-config any-owner goal

repetition

Stop the game when a position is repeated.

```
repetition
```

Remarks

The **repetition** goal is True if a position is exactly repeated a third time with the same player on the move. You must use repetition with one of the goals: draw-condition, win-condition, loss-condition.

Zillions still checks for draw by repetition if the **repetition** keyword is not used in the game; using **repetition** merely allows you to convert that result to a win or a loss.

When comparing positions to see if a repetition has occurred, the entire state of the board is noted, including the current piece attributes. Two pieces of the same color, piece-type, and attributes are considered identical here just as in the game's movement rules, i.e. it does not matter if two identical pieces have swapped places, the position is still considered a repetition. The off-board stores are currently not considered when comparing positions.

Like checkmated, stalemated, and captured, **repetition** must be at the top goal level.

Usage

```
(game
...
  (draw-condition (White Black)
    repetition ; if the position repeats, well call that a draw
  )
...
)
```

Examples

```
; Its a draw if either side repeats the position
(draw-condition (White Black) repetition)

; The Green side loses if he repeats the position
(loss-condition (Green) repetition)
```

See Also

goal

set-attribute

Changes an attribute of a piece when an add is done..

(set-attribute <attribute> <condition>)

Remarks

set-attribute affects the piece that is on the current square after the move is made. <attribute> is an attribute of a piece defined with the attribute construct.

A single add should not attempt to set the attribute of a piece to two different values.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        ; Set never-moved? attribute
        (set-attribute never-moved? true)
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
; Sets the pieces "my-attribute" attribute to True
(set-attribute my-attribute? true)

; Sets the pieces "my-attribute" attribute to True if its our piece,
; or False if its an opponents
(set-attribute my-attribute? friend?)

; If the piece is a King of ours, set the "my-attribute" to True,
; otherwise set it to False
(set-attribute my-attribute? (and friend? (piece? King)))
```

See Also

attribute <condition>

set-flag

Set's a flag's value.

(set-flag <flag-name> <condition>)

Remarks

A flag is a Boolean (true or false) variable that can be used within a move generation block to stand for anything you want. Use **set-flag** to set a flags value. A flag may be tested using flag? and not-flag? in a condition.

It is not necessary to explicitly "define" a flag anywhere. Just using a flag is enough to inform Zillions of its existence. A flag has no default, so remember to use **set-flag** to initialize a flag before testing it.

Flags are intended to be used only within a move generation block, rather than across them. Unlike position flags flags are independent of board position.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        ; Set my-flag to true if a1 is empty, false otherwise
        (set-flag my-flag (empty? a1))
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
; Sets the flag "my-flag" to true
(set-flag my-flag true)

; If the piece is a King of ours, set "my-flag" to true,
; otherwise set it to false
```

```
(set-flag my-flag (and friend? (piece? King)))
```

See Also

flag? not-flag? set-position flag <condition>

set-position-flag

Set a position flags value.

```
(set-position-flag <flag-name> <condition>)
(set-position-flag <flag-name> <condition> <position>)
(set-position-flag <flag-name> <condition> <direction>)
```

Remarks

To test a position flags value use position-flag? and not-position-flag?

Position flags are true/false (Boolean) variables associated with positions. The flags for each position are initialized to false at the start of a move generation block for a piece. They only retain their values during move generation while the move code inside that block is executed.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        ; Set my-flag for the current position to
        ; true if a1 is empty, false otherwise
        (set-position-flag my-flag (empty? a1))
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
; Set the flag "my-flag" for the current position to false
(set-position-flag my-flag false)

; Set the flag "enemy-on-position" for the current position
; to be true if and only if there is an enemy there
(set-position-flag enemy-on-position enemy?)

; Set the flag "enemy-on-position" for the position in the
```

```
; n direction to be true if and only if there is an enemy there
(set-position-flag enemy-on-position (enemy? n) n)

; Set "my-empty-flag" for the current position to true if
; a1 is empty, false otherwise
(set-position-flag my-empty-flag (empty? a1))

; Set "my-empty-flag" for a1 to true if a1 is empty, false
; otherwise
(set-position-flag my-empty-flag (empty? a1) a1)
```

See Also

position-flag? not-position-flag? <condition> set-flag?

solution

Specify the Zillions Save Game (*.zsg) file that has a solution for the current puzzle..

```
(solution <file-name>)
```

Remarks

This specified file will be loaded whenever the user selects Help - Solution. If no **solution** statement is used, the Help - Solution menu item will remain grayed out.

Usage

```
(game
...
  (solution "Solutions\TurnOff\3Medium2.zsg")
...
)
```

Examples

None.

See Also

None.

-sound capture-sound, change-sound, click-sound, draw-sound, drop-sound, loss-sound, move-sound, opening-sound, release-sound, win-sound

Sets a sound file to be played.

```
(capture-sound <file-string>)
(change-sound <file-string>)
(click-sound <file-string>)
(draw-sound <file-string>)
(drop-sound <file-string>)
(loss-sound <file-string>)
(move-sound <file-string>)
(opening-sound <file-string>)
(release-sound <file-string>)
(win-sound <file-string>)
```

Remarks

These keywords allow you to specify a ".wav" file to be played at certain times:

KEYWORD	DEFAULT	Played
capture-sound	Capture.wav	When a piece is captured
change-sound	Change.wav	When a piece is changed from one type to another
click-sound		When a piece is clicked on (to start a drag)
draw-sound	Draw.wav	When a draw occurs in the game
drop-sound	Drop.wav	When a piece is dropped onto the board
loss-sound	Loss.wav	When the player loses the game
move-sound	Move.wav	When the opponent moves a piece
opening-sound		When a new game begins
release-sound		When the player releases a dragged piece
win-sound	Win.wav	When the player wins the game

Sounds are only active when "Sounds" is turned on in the "Options..." window. You can use programs like the Multimedia Recorder to create your own sounds and add them to Zillions games.

Usage

```
(game
```

```
...
  (drop-sound "splat.wav")
...
)
```

Examples

```
; Plays the "aarrggh.wav" sound when a piece is captured
(capture-sound "aarrggh.wav")
```

See Also

music

stalemated

Stop the game when a side can't move..

stalemated

Remarks

The **stalemated** goal is True when the player to move has no legal moves and is not allowed to pass either. The right to pass a turn is determined by the "pass turn" option. If unspecified, the default is for a stalemate to be a loss in a single-player game, or a draw otherwise. You must use **stalemated** with one of the goals: draw-condition, win-condition, loss-condition.

Like checkmated, captured, and repetition, stalemated must be at the top goal level.

Usage

```
(game
...
  (draw-condition (White Black)
    stalemated ; if a side cant move, its a draw (Chess)
  )
...
)
```

Examples

```
; Its a draw if either side is left without a move
(draw-condition (X O) stalemated)

; The Chaos side wins if he has no moves left
(win-condition (Chaos) stalemated)

; The Order side loses if he has no moves left
(loss-condition (Order) stalemated
```

See Also

option checkmated goal

strategy

Sets the strategy descriptions for a game variant.

```
(strategy <text>)
```

Remarks

strategy defines what the user sees when he asks for strategy of the current game under Help, Strategy. Pass a string.

Default is no description, so if you want the user to know strategy for this game, make sure you use **strategy**.

Usage

```
(game
...
   (strategy "Whenever possible, confuse your opponent by gargling loudly.")
...
)
(variant
...
  (strategy "Look out for the Rooks! They can be tricky.")
...
)
```

Examples

```
(strategy
  "The best moves are near the center, where mobility is best."
)
```

See Also

description history

symmetry

Defines a symmetry on the board.

```
(symmetry <player> (<direction> <direction>)...(<direction> <direction>))
```

Remarks

symmetry allows you to define that different sides are moving in different directions, e.g. in Chess, Blacks Pawns move the opposite direction as Whites Pawns. The symmetries defined are one-way and apply to all pieces and goals in the game.

Usage

```
(game
...
  (board
  ...
    (symmetry Red (nw sw)(sw nw) (ne se)(se ne))
  ...
  )
...
)
```

Examples

```
; All "n" directions are converted into "s" directions for Black
(symmetry Black (n s))

; If the directions are defined normally, this essentially flips the
; map upside down for Black
(symmetry Black (n s)(s n) (ne se)(se ne) (nw sw)(sw nw))

; For orthogonal directions, Green is playing rotated 90 degrees from other playe
(symmetry Green (e n) (n w) (w s) (s e))
```

See Also

board

title

Gives the game variant a title.

```
(title <string>)
```

Remarks

The title specifies a name that may be shown to the user. For a game or variant, the title is shown as a menu item and, when once it is selected, also appears in the windows title bar. A variant with the title of a dash:

(variant (**title** "-"))

will show up on the Variant menu as a separator line. This is good for organizing lots of variants of different types, such as variants on different boards.

Usage

```
Usage:

(game
...
  (title "Chess")
...
)
```

Examples

```
(title "My Game") ; Sets the title of the game to "My Game"
```

See Also

game variant

total-piece-count

Stop the game when there is a certain number of pieces on the board.

(total-piece-count <number>)
(total-piece-count <number> <piece-type>) ***version 2.0+ feature only***

Remarks

A goal that evaluates to true when the total number of pieces on the board reaches the specified number. All pieces are counted, regardless of who owns them. To total only one side's pieces, use pieces-remaining instead. Neutral pieces are counted too, which is nice for games like Blobs, where you want to know when the board is filled, but neutral pieces are used for walls.

Unlike other goals it's possible to omit the player list when using **total-piece-count** at the top level of a win/loss/draw-condition. When used in this way, the winner is assumed to be the side that just moved.

When a <piece-type> argument is used, only pieces of that type are totalled. Otherwise, all pieces are totalled.

Usage

```
(game
...
  (count-condition (total-piece-count 64))
...
)
```

Examples

```
; When there are exactly 64 pieces on the board, the game is over and
; the side with most pieces wins
(count-condition (total-piece-count 64))

; When there are exactly 64 pieces on the board, the game is over and
; the player that last moved wins
(win-condition (total-piece-count 64)) ; note: omits the player list

; The player who captures the last prize wins
(win-condition (total-piece-count 0 prize)) ; note: omits the player list

; White wins immediately if there are exactly 12 pieces on the board
(win-condition (White) (total-piece-count 12))
```

See Also

pieces-remaining <goal>

translate

Define translations for words in a ZRF.

(translate <translation>...<translation>) ***version 1.1.1+ feature only***

Remarks

translate allows you to add a set of spoken language translations to an existing ZRF file that affect every variant in the ZRF. The translations only affect text shown to the user. Since saved games are stored using untranslated strings, users with different translations may share saved games. Similarly, users with different translations may still play each other over networks. You can view the moves before translation by selecting the "Verbose notation in moves list" option in Authoring mode.

Only the names of variants, players, pieces, and positions are translated. Piece descriptions and game descriptions/histories/strategies are unaffected by **translate**.

The translations always take place; if you took your English ZRF and put German translations at the beginning, it would now be a German ZRF. Thus, separate files must be maintained for each language, though the maintenance of them is easier. (See the example below.)

Each <translation> takes the following form:

(<original-string> <translated-string>)

translate blocks should be placed at the outer-most level of a ZRF, like game and variant blocks.

Usage

```
(translate
  ("White" "Weiß")
  ("Black" "Schwarz")
  ("Pawn" "Bauer")
  ("Knight" "Springer")
  ("Bishop" "Läufer")
  ("Rook" "Turm")
  ("Queen" "Dame")
  ("King" "König")
  ("Chess" "Schach")
)
```

Examples

```
; BritishVersion.zrf
```

```
(translate ("Color" "Colour"))
(include "AmericanVersion.zrf") ; Rest of ZRF is the same
```

See Also

None.

turn-order

Sets the order the players move in.

```
(turn-order <turn>...<turn>)
```

Remarks

In its most basic form **turn-order** allows you to specify the games turn sequence: who plays first, who plays next, and so on. When the end of the list arrives, Zillions starts the turn sequence over at the beginning, unless there is a **repeat**, which modifies this behavior. **turn-order** can also place restrictions on the type of move that may be played and allow one side to move another sides pieces. Each <turn> argument may be one of the following:

<player>	The given player plays this turn
(<player> <move-type>)	The given player plays this turn, but only moves of the given type
(<player> <player>)	First player specified moves second players pieces
(<player> <player> <move-type>)	First player makes this type of move with second players pieces
repeat	At the end of the turn order go back to here and repeat

Only players that show up as having a turn in the turn order are playable by the computer or human. Other players are considered "neutral" sides and will not show up in the "Choose Sides" dialog.

Note that if you need neutral pieces automatically moved around in between the moves of real players, it is usually better to have real players move these pieces around rather than making up additional "fake" players. For example:

```
 (turn-order White (Black Neutral1) Black (Black Neutral2))
```

The reason is that the fewer players with moves, the better Zillions can calculate; there is a lot of complexity in searching moves for games with more than two players.

Usage

```
(game
...
  (turn-order White Black)
...
)
```

Examples

```
; The sequence is White, Black, White, Black, etc.
(turn-order White Black)

; The sequence is White, White, Black, White, Black, etc.
(turn-order White repeat White Black)

; White, Black, White, Black, etc.
; but Whites first move must be of type "capture-move"
(turn-order (White capture-move) repeat Black White)

; Allows each side a move of type "capture-move" after their normal move
(turn-order White (White capture-move) Black (Black capture-move))

; After each normal move the moving side gets to move one
; of Neutral-Sides pieces
(turn-order White (White Neutral-Side) Black (Black Neutral-Side))

; Whites first move is a "capture-move" type move of one
; of Neutral-Sides pieces
(turn-order (White Neutral-Side capture-move) repeat ...)
```

See Also

move-type

unlink

Unlinks positions.

(unlink <unlink-arg>...<unlink-arg>)

Remarks

unlink accepts a list of any number of arguments <unlink-arg>, where each argument is in one of the following 3 forms:

<Position>	Deletes all links (if any) going into and out of that position
(<Position> <Position>)	Deletes all links (if any) directly connecting the positions
(<Position> <Direction>)	Deletes the link in the given direction from the position, if it exists

Usage

```
(game
...
  (board
  ...
    (unlink a b (c d) e f)
  ...
  )
...
)
```

Examples

```
(unlink e4 d7 c6) ; Removes all links to/from positions e4, d7, c6
(unlink (a1 b1) ) ; Removes all links connecting a1 to b1
(unlink (a1 b1) (a8 b8)) ; Disconnects a1 and b1, Disconnects a8 and b8

; Removes the "n" link coming out of position a1
; Note: this has no effect on any links coming into a1.
(unlink (a1 n))

; The different types may all appear on the same list
(unlink e4 (a1 a2) d7 (a1 ne))
```

See Also

kill-positions board

verify

Test whether a condition is True.

(verify <condition>)

Remarks

Use **verify** to test for a certain condition, such as if a certain piece type is present or a position is empty.

If the condition is False, then generation in that move block stops immediately. Note that you can use (**verify** false) to force a stop if you know no more moves can be generated in that move block.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      (
      ...
        (verify (empty? n))
      ...
      )
    ...
    )
  ...
  )
...
)
```

Examples

```
(verify false) ; Stop generating moves
(verify (and friend? (piece? King))) ; Continue if at a friendly King
```

See Also

<condition>

while

Execute move language instructions repeatedly.

(while <condition> <instruction>...<instruction>)

Remarks

The enclosed instruction(s) are executed as long as the given condition holds true. When **while** fails, the move generation proceeds to the next statement for this piece.

Usage

```
(game
...
  (piece
  ...
    (moves
    ...
      ; Add a move to the furthest square in the "n" direction
      ((while (on-board? n) n) add)
    ...
    )
  ...
  )
...
)
```

Examples

```
; While the current position is empty, add a move and go in the "n" direction
(while empty? add n)
```

See Also

<condition> <instruction>

win-condition, loss-condition, draw-condition

Defines when the game is over and who wins.

```
(win-condition (<player>...<player>) <goal>)
(loss-condition (<player>...<player>) <goal>)
(draw-condition (<player>...<player>) <goal>)
```

Remarks

An end-condition is a rule indicating when the game is over. The three keywords indicate three different results:

- **win-condition**: the player wins
- **loss-condition**: the player loses
- **draw-condition**: the player ties

One or more players must be specified, with the exception of the total-piece-count goal, which is not based on a side and thus can use an alternate syntax. The goal will be checked for each player given after every move of the game.

To determine the winner by counting pieces after a goal has been reached, use count-condition instead.

Usage

```
(game
...
  (loss-condition (White Black)
   ; if either Black or White lose their King, they lose the game
   (captured King)
  )
...
)
```

Examples

```
; The game is a draw if White is stalemated
(draw-condition (White) stalemated)

; If either White or Black loses his King, he loses
(loss-condition (White Black) (captured King)
```

See Also

<goal> count-condition total-piece-count

zone

Groups positions into a zone.

```
(zone <zone-arg>...<zone-arg>)
```

Remarks

The **zone** keyword allows you define a "zone," an arbitrary set of positions that can be treated as a single unit. Zones are an easy way to define groups of positions where special actions need to occur, like the promotion of pieces. The <zone-arg> arguments are as follows:

(name <zone-name>)	Refer to the zone by this name
(players <player>...<player>)	This zone is for these player(s)
(positions <position>...<position>)	The set of positions that comprise the zone

For a **zone** to be useful, it must have all three of these arguments.

One feature of zones is that a zone may contain different positions for different players. For example, in Chess we might define a Pawn promotion zone in the board construct as follows:

```
(zone
  (name promotion-zone)
  (players White)
  (positions a8 b8 c8 d8 e8 f8 g8 h8)
)
(zone
  (name promotion-zone)
  (players Black)
  (positions a1 b1 c1 d1 e1 f1 g1 h1)
)
```

Then in the Pawns move logic we could write:

```
(if (in-zone? promotion-zone) (add Knight Bishop Rook Queen))
```

meaning "if the Pawn is in the promotion zone then promote it to a Knight, Bishop, Rook, or Queen." The positions of "promotion-zone" are defined relative to each side so the same Pawn and movement logic will work regardless of which side owns the pawn.

Up to 32 zones may be defined.

Usage

```
(game
...
```

```
  (board
  ...
    (zone
      (name Center)
      (players White Black)
      (positions e4 e5 d4 d5)
    )
  )
...
)
```

Examples

None.

See Also

name players positions board