Games: Fun Examples

Version 4.1.3

November 20, 2008

The `PLT Games` executable (or `plt-games` under Unix) lets you select one of the games distributed by PLT or other games installed as sub-collections of the `"games"` collection (see §2 “Implementing New Games”).

Contents

1 Bundled Games

1.1  Aces — Solitaire Card Game

To play Aces, run the PLT Games program. (Under Unix, it's called plt-games).

Aces is a solitaire card game. The object is to remove all of the cards from the board, except the four Aces.

Remove a card by clicking it. You may remove a card when two conditions are true. First, it must be at the bottom of one of the four stacks of cards. Second, either the ace of the same suit, or a higher card of the same suit must also be at the bottom of one of the four stacks of cards.

You may also move any card from the bottom of one of the stacks to an empty stack by clicking it. If there are still cards in the deck on the right, you may click the deck to deal four new cards, one onto the bottom of each stack.

Good Luck!

1.2  Go Fish — Kid's Card Game

To play Go Fish, run the PLT Games program. (Under Unix, it's called plt-games).

Go Fish is the children's card game where you try to get rid of all you cards by forming pairs. You play against two computer players.

On each turn, if you have a match in your hand, drag one of the matching cards to your numbered box, and the match will move into the box.

After forming matches from your own hand, drag one of your cards to an opponent's area to ask the opponent for a matching card:

- If the opponent has a card with the same value as the card that you drag, the opponent will give you the card, and they'll go into your match area. Drag another card to an opponent.
- If the opponent has no matching card, the top card on draw pile will move, indicating that you must "Go Fish!" Draw a card by dragging it from the draw pile to your hand. If the drawn card gives you a match, then the match will automatically move into your match area, and it's still your turn (so drag another card to one of the opponents).

The game is over when one player runs out of cards. The winner is the one with the most matches.

The status line at the bottom of the window provides instructions as you go. The computer players are not particularly smart.

1.3  Crazy 8s — Card Game

To play Crazy 8s, run the PLT Games program. (Under Unix, it's called `plt-games`).

Try to get rid of all you cards by matching the value or suit of the top card in the discard pile. In the default mode, click a card to discard it; you can adjust the options so that you discard by dragging a card from your hand to the discard pile.

An 8 can be discarded at any time, and in that case, the player who discarded the 8 gets to pick any suit for it (hence the craziness of 8s). When you discard an 8, a panel of buttons appears to the right of the discard pile, so you can pick the suit.

A player can choose to draw a card instead of discarding, as long as cards are left in the draw pile. A player's turn continues after drawing, so a player can continue drawing to find something to discard. In the default mode, click the face-down draw pile in the middle of the table; you can adjust the options to that you draw by dragging it from the draw pile to your hand.

If no cards are left in the deck, a player may pass instead of discarding. To pass, click the Pass button.

The status line at the bottom of the window provides instructions as you go.

1.4  Blackjack — 21 Card Game

To play Blackjack, run the PLT Games program. (Under Unix, it's called `plt-games`).

Standard Blackjack rules with the following specifics:

- 1 player (not counting the dealer).
- 4 decks, reshuffled after 3/4 of the cards are used.
- Dealer stands on soft 17s.
- Splitting is allowed only on the first two cards, and only if they are equal. 10 and the face cards are all considered equal for splitting.
- Doubling is allowed on all unsplit hands, not on split hands.
- No blackjacks after splitting.

- No surrender.
- No insurance.
- No maximum under-21 hand size.
- Dealer's second card is not revealed if the player busts (or both halves of a split hand bust).

1.5  Rummy — Card Game

To play Rummy, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

This is a simple variant of Rummy.

Put all cards in your hand into straights (3 or more cards in the same suit) and 3- or 4-of-a-kind sets to win. Each card counts for only one set. Aces can be used in both A-2-3 sequences and Q-K-A sequences.

When all of your cards fit into sets (the game detects this automatically), you win.

On each turn, you can either draw from the deck or from the top of the discard pile (drag from either to your hand), then you must discard one of your own cards (by dragging from your hand to the discard pile).

The status line at the bottom of the window provides instructions as you go. The computer player is fairly smart.

1.6  Spider — Solitaire Card Game

To play Spider, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Spider is a solitaire card game played with 104 cards. The cards can include either a single suit, two suits, or four suites. (Choose your variant through the Options item in the Edit menu.)

Terminology:

- *Tableau*: one of the ten stacks of cards in the play area. The game starts with six cards in the first four tableaus, and five cards in the rest; only the topmost card is face up, and others are revealed when they become the topmost card of the tableau.
- *Sequence*: a group of cards on the top of a tableau that are in the same suit, and that are in sequence, with the lowest numbered card topmost (i.e., closer to the bottom of the screen). King is high and ace is low.

The object of the game is to create a sequence with ace through king, at which point the sequence is removed from play. Create eight such sequences to win the game.

On each move, you can take one of the following actions:

- Move a sequence from any tableau to one whose topmost card (i.e., closest to the bottom of the screen) has a value that's one more than the sequence's value. Note that if the top card of the target tableau has the same suit as the sequence, a larger sequence is formed, but the target tableau's card is not required to have the same suit.
- Move a sequence to an empty tableau.
- Deal ten cards from the deck (in the upper left corder), one to each tableau. This move is allowed only if no tableau is empty.

To move a sequence, either drag it to the target tableau, or click the sequence and then click the top card of the target tableau (or the place where a single card would be for an empty tableau). Click a select card to de-select it. Clicking a card that is not a valid target for the currently selected sequence causes the clicked card's sequence to be selected (if the card is face up in a sequence).

To deal, click the deck.

To undo a move, use Undo from the Edit menu.

1.7  Memory — Kid's Game

To play Memory, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Flip two cards in a row that have the same picture, and the cards are removed. If the two cards don't match, they are flipped back over, and you try again. Each card has a single match on the board. The game is over and the clock stops when all cards are removed.

1.8  Slidey — Picture Puzzle

To play Slidey, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Click a tile to slide it into the adjacent space, and keep shifting tiles that way to repair the picture.

1.9  Same — Dot-Removing Game

To play Same, run the PLT Games program. (Under Unix, it's called plt-games).

The object of Same is to score points by removing dots from the board. To remove a dot, click on it. As long as there is another dot of the same color next to the clicked dot, it will disappear along with all adjacent dots of the same color. After the dots disappear, dots in the rows above the deleted dots will fall into the vacated spaces. If an entire column is wiped out, all of the dots from the right will slide left to take up the empty column's space.

Your score increases for each ball removed from the board. The score for each click is a function of the number of balls that disappeared. The This Click label shows how many points you would score for clicking the dots underneath the mouse pointer. The score varies quadratically with the number of balls, so eliminating many balls with one click is advantageous.

Click the New Game button to play again.

1.10  Minesweeper — Logic Game

To play Minesweeper, run the PLT Games program. (Under Unix, it's called plt-games).

Remove all the tiles that have no bomb underneath. When you remove such a tile, a number appears that indicates how many of the surrounding squares (up to 8) have a bomb; a blank means zero bombs, and the game automatically uncovers all surrounding tiles in that case.

Right- or Control-click to flag a tile that you think has a bomb, so that you cannot accidentally uncover it. Right- or Control-click again to remove the flag.

You don't have to use flags. When all of the non-bomb tiles are removed, the game is over, and the clock stops.

1.11  Paint By Numbers — Logic Game

To play Paint By Numbers, run the PLT Games program. (Under Unix, it's called plt-games).

The object of Paint By Numbers is to discover which cells should be colored blue and which should be colored white. Initially, all squares are grey, indicating that the correct colors are not known. The lists of numbers to the left and above the grid are your clues to the correct color of each square. Each list of numbers specifies the pattern of blue squares in the row beside it or the column below it. Each number indicates the length of a group of blue squares. For example, if the list of numbers beside the first row is 2 3 then you know that there is a contiguous block of two blue squares followed by a contiguous block of three blue squares

with at least one white square between them. The label does not tell you where the blue squares are, only their shapes. The trick is to gather as much information as you can about each row, and then use that information to determine more about each column. Eventually you should be able to fill in the entire puzzle.

Click on a square to toggle it between blue and gray. Hold down a modifier key (shift, command, meta, or alt depending on the platform) to toggle a square between white and gray. The third button under unix and the right button under windows also toggles between white and gray.

For some puzzles, hints are available. Choose the Nongram|Show Mistakes menu item to receive the hints. This will turn all incorrectly colored squares red.

Thanks to Shoichiro Hattori for his puzzles! Visit him on the web at:

`http://hattori.m78.com/puzzle/`

Thanks also to many of the contributors to the Kajitani web site for permission to redistribute their puzzles. Visit them online at:

`http://nonogram.freehostia.com/pbn/index.html`

The specific contributers who have permitted their puzzles to be redistributed are:

snordmey /at/ dayton <dot> net
jtraub /at/ dragoncat <dot> net
e0gb258s /at/ mail <dot> erin <dot> utoronto <dot> ca
mattingly /at/ bigfoot <dot> com
jennifer <dot> forman /at/ umb <dot> edu
karen <dot> hoover /at/ bigfoot <dot> com
sssstree /at/ ix <dot> netcom <dot> com
we_bakers_3 /at/ earthlink <dot> net
bbart /at/ cs <dot> sfu <dot> ca
jonesjk /at/ thegrid <dot> net
rrichard /at/ lexitech <dot> ca
helena <dot> montauban /at/ auroraenergy <dot> com <dot> au
barblane /at/ ionsys <dot> com
m5rammy /at/ maale5 <dot> com
nmbauer /at/ sprynet <dot> com
ncfrench /at/ aol <dot> com
km29 /at/ drexel <dot> edu
jjl /at/ stanford <dot> edu
disneyfan13 /at/ hotmail <dot> com
richard /at/ condor-post <dot> com
lady_tabitha /at/ yahoo <dot> com
vaa /at/ psulias <dot> psu <dot> edu
kimbhall /at/ yahoo <dot> com

```
kcottam /at/ cusa <dot> com
karganov /at/ hotmail <dot> com
jdmaynard /at/ excite <dot> com
mnemoy /at/ gameworks <dot> com
arrelless /at/ jayco <dot> net
azisi /at/ skiathos <dot> physics <dot> auth <dot> gr
whoaleo /at/ hotmail <dot> com
tucker1999 /at/ earthlink <dot> net
bergles /at/ yahoo <dot> com
elisabeth <dot> springfelter /at/ lanab <dot> amv <dot> se
ewhaynes /at/ mit <dot> edu
mjcarroll /at/ ccnmail <dot> com
dahu /at/ netcourrier <dot> com
joy /at/ dcs <dot> gla <dot> ac <dot> uk
piobst /at/ wam <dot> umd <dot> edu
dani681 /at/ aol <dot> com
Talzhemir <pixel /at/ realtime <dot> net>
hkittredge /at/ hotmail <dot> com
allraft /at/ sccoast <dot> net
karlvonl /at/ geocities <dot> com
ailsa /at/ worldonline <dot> nl
Carey Willis <N8NRG /at/ hotmail <dot> com>
citragreen /at/ hotmail <dot> com
dhalayko /at/ cgocable <dot> net
jontive1 /at/ elp <dot> rr <dot> com
hublan /at/ rocketmail <dot> com
barbridgway /at/ compuserve <dot> com
mijoy /at/ mailcity <dot> com
joostdh /at/ sci <dot> kun <dot> nl
gossamer_kwaj /at/ hotmail <dot> com
williamson /at/ proaxis <dot> com
vacko_6 /at/ hotmail <dot> com
jojess /at/ earthlink <dot> net
```

1.12 Lights Out — Logic Game

To play Lights Out, run the PLT Games program. (Under Unix, it's called plt-games).

The object of this game is to turn all of the lights off. Click on a button to turn that light off, but beware it will also toggle the lights above, below to the left and to the right of that button.

Good luck.

1.13 Pousse — Tic-Tac-Toe-like Game

To play Pousse, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Pousse (French for "push", pronounced "poo-ss") is a 2 person game, played on an N by N board (usually 4 by 4). Initially the board is empty, and the players take turns inserting one marker of their color (X or O) on the board. The color X always goes first. The columns and rows are numbered from 1 to N, starting from the top left, as in:

```
   1 2 3 4
  +-+-+-+-+
1 | | | | |
  +-+-+-+-+
2 | | | | |
  +-+-+-+-+
3 | | | | |
  +-+-+-+-+
4 | | | | |
  +-+-+-+-+
```

A marker can only be inserted on the board by sliding it onto a particular row from the left or from the right, or onto a particular column from the top or from the bottom. So there are 4*N possible "moves" (ways to insert a marker). They are named Li, Ri, Ti, and Bi respectively, where i is the number of the row or column where the insertion takes place.

When a marker is inserted, there may be a marker on the square where the insertion takes place. In this case, all markers on the insertion row or column from the insertion square upto the first empty square are moved one square further to make room for the inserted marker. Note that the last marker of the row or column will be pushed off the board (and must be removed from play) if there are no empty squares on the insertion row or column.

A row or a column is a *straight* of a given color if it contains N markers of the given color.

The game ends either when an insertion

- repeats a previous configuration of the board; in this case the player who inserted the marker LOSES.
- creates a configuration with more straights of one color than straights of the other color; the player whose color is dominant (in number of straights) WINS.

A game always leads to a win by one of the two players. Draws are impossible.

This game is from the 1998 ICFP programming contest.

1.14  Gobblet — Strategy Game

To play Gobblet, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Gobblet! is a board game from Blue Orange Games:

`http://www.blueorangegames.com/`

Our 3x3 version actually corresponds to **Gobblet! Jr.**, while the 4x4 version matches Gobblet!.

The Blue Orange web site provides rules for **Gobblet! Jr.** and **Gobblet!**. The rules below are in our own words; see also the Blue Orange version.

1.14.1 Game Rules

The 3x3 game is a generalization of tic-tac-toe:

- The object of the game is to get three in a row of your color, vertically, horizontally, or diagonally. Size doesn't matter for determining a winner.
- Each player (red or yellow) starts with 6 pieces: two large, two medium, and two small.
- On each turn, a player can either place a new piece on the board, or move a piece already on the board—from anywhere to anywhere, as long as the "from" and "to" are different.
- A piece can be placed (or moved to) an empty space, or it can be placed/moved on top of a smaller piece already on the board, "gobbling" the smaller piece. The smaller piece does not have to be an opponent's piece, and the smaller piece may itself have gobbled another piece previously.
- Only visible pieces can be moved, and only visible pieces count toward winning. Gobbled pieces stay on the board, however, and when a piece is moved, any piece that it gobbled stays put and becomes visible.
- If moving a piece exposes a winning sequence for the opponent, and if the destination for the move does not cover up one of the other pieces in the sequence, then the opponent wins—even if the move makes a winning sequence for the moving player.
- Technically, if a player touches a piece, then the piece must be moved on that turn. In other words, you're not allowed to peek under a piece to remind yourself whether it gobbled anything. If the piece can't be moved, the player forfeits. This particular rule is not enforced by our version — in part because our version supports a rewind button, which is also not in the official game.

The 4x4 game has a few changes:

- The object of the game is to get four in a row of your color.
- Each player (red or yellow) starts with 12 pieces: three large, three medium-large, three medium-small, and three small.
- Each player's pieces are initially arranged into three stacks off the board, and only visible pieces can be moved onto the board. The initial stacks prevent playing a smaller piece before a corresponding larger piece.
- When a piece is moved from off-board onto the board, it must be moved to either (1) an empty space, or (2) a space to gobble an opponent's piece that is part of three in a row (for the opponent). In other words, a new piece can gobble only an opponent's piece, and only to prevent an immediate win on the opponent's next turn. These restrictions do not apply when a piece that is already on the board is moved.

1.14.2 Controls

Click and drag pieces in the obvious way to take a turn. The shadow under a piece shows where it will land when you drop it.

Use the arrow keys on your keyboard to rotate the board. Use the - and = keys to zoom in and out. Use _ and + to make the game smaller and larger. (Changing the size adjusts perspective in a slightly different way than zooming.) Depending on how keyboard focus works on your machine, you may have to click the board area to make these controls work.

The button labeled < at the bottom of the window rewinds the game by one turn. The button labeled > re-plays one turn in a rewound game. An alternate move can be made at any point in a rewound game, replacing the old game from that point on.

1.14.3 Auto-Play

Turn on a computer player at any time by checking the Auto-Play Red or Auto-Play Yellow checkbox. If you rewind the game, you can choose an alternate move for yourself or for the auto-player to find out what would have happened. The auto-player is not always deterministic, so replying the same move might lead to a different result. You can disable an auto-player at any point by unchecking the corresponding Auto-Play" checkbox.

Important: In the 3x3 game, you *cannot* win as yellow against the smart auto-player (if the auto-player is allowed to play red from the start of the game). In other words, red has a forced win in the 3x3 game, and the smart auto-player knows the path to victory. You might have a chance to beat the red player in the default mode, though, which is represented by the Ok choice (instead of Smart) in the Auto-Play Options dialog.

Configure the auto-player by clicking the Auto-Play Options button. Currently, there's no difference between Smart and Ok in the 4x4 game.

1.15  Jewel — 3-D Skill Game

To play Jewel, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

The board is an 8 by 8 array of jewels of 7 types. You need to get 3 or more in a row horizontally or vertically in order to score points. You can swap any two jewels that are next to each other up and down or left and right. The mechanic is to either:

- Click the mouse on the first one, then drag in the direction for the swap.
- Move a bubble using the arrow keys, lock the bubble to a jewel with the space bar, and the swap the locked jewel with another by using the arrow keys. Space unlocks a locked bubble without swapping.

Jewels can only be swapped if after the swap there are at least 3 or more same shape or color in a row or column. Otherwise the jewels return to their original position. There is a clock shown on the left. When it counts down to 0 the game is over. Getting 3 in a row adds time to the clock.

Hit spacebar to start a new game then select the difficulty number by pressing 0, 1, 2, 3, or 0. You can always press ESC to exit. During playing press P to pause the game.

The code is released under the LGPL. The code is a conversion of Dave Ashley's C program to Scheme with some modifications and enhancements.

Enjoy.

1.16  Parcheesi — Board Game

To play Parcheesi, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

Parcheesi is a race game for four players. The goal is for each player to move their pieces from the starting position (the circles in the corners) to the home square (in the center of the board), passing a nearly complete loop around the board in the counter-clockwise direction and then heads up towards the main row. For example, the green player enters from the bottom right, travels around the board on the light blue squares, passing each of the corners, until it reaches the middle of the bottom of the board, where it turns off the light blue squares and heads into the central region.

On each turn, the player rolls two dice and advances the pawn, based on the die rolls. Typically the players may move a pawn for each die. The pawn moves by the number of pips showing on the die and all of the dice must be used to complete a turn.

There are some exceptions, however:

- You must roll a 5 (either directly or via summing) to enter from the start area to the main ring.
- If two pieces of the same color occupy a square, no pieces may pass that square.
- If an opponent's piece lands on your piece, you piece is returned to the starting area and the opponent receives a bonus of 20 (which is treated just as if they had rolled a 20 on the dice).
- If your piece makes it home (and it must do so by exact count) you get a bonus of 10, to be used as an additional die roll.

These rules induce a number of unexpected corner cases, but the GUI only lets you make legal moves. Watch the space along the bottom of the board for reasons why a move is illegal or why you have not used all of your die rolls.

The automated players are:

- Reckless Renee, who tries to maximize the chances that someone else bops her.
- Polite Polly, who tries to minimize the distance her pawns move. ("No, after *you*. I insist.")
- Amazing Grace, who tries to minimize the chance she gets bopped while moving as far as possible.

1.17  Checkers — Board Game

To play Checkers, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

This simple checkers game (with no AI player) is intended as a demonstration use of the `games/gl-board-game` library.

1.18  Chat Noir — Puzzle Game

To play Chat Noir, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

The goal of the game is to stop the cat from escaping the board. Each turn you click on a circle, which prevents the cat from stepping on that space, and the cat responds by taking a

step. If the cat is completely boxed in and thus unable reach the border, you win. If the cat does reach the border, you lose.

The game was inspired by this one the one at Game Design and has essentailly the same rules.

This game is written in the How to Design Programs Intermediate language. It is a model solution to the final project for the introductory programming course at the University of Chicago in the fall of 2008, as below.

```
; ;;constants
(define circle-radius 20)
(define circle-spacing 22)

; ;;datadefinitions

; ;;aworldis:
; ;;(make-worldboardposnstatenumber)
(define-struct world (board cat state size))

; ;;astateiseither:
; ;;-'playing
; ;;-'cat-won
; ;;-'cat-lost

; ;;aboardis
; ;;(listofcell)

; ;;acellis
; ;;(make-cell(make-posnint[0-board-size]
; ;;int[0-board-size])
; ;;boolean)
(define-struct cell (p blocked?))


; ;
; ;
; ;
; ;
; ;
; ;;;;;;;
; ;;;;;;;;;;
; ;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;
; ;;;;;;;
; ;


; ;;world->image:world->image
(define (world->image w)
  (chop-whiskers
   (overlay (board->image (world-board w) (world-size w))
            (move-pinhole
             (cond
               [(equal? (world-state w) 'cat-won) happy-cat]
               [(equal? (world-state w) 'cat-lost) sad-cat]
               [else thinking-cat])
             (- (cell-center-x (world-cat w)))
             (- (cell-center-y (world-cat w)))))))

(check-expect
 (world->image
  (make-world (list (make-cell (make-posn 0 1) false))
              (make-posn 0 1)
              'playing
              2))
 (overlay
  (board->image (list (make-cell (make-posn 0 1) false))
                2)
  (move-pinhole thinking-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))
(check-expect
 (world->image
  (make-world (list (make-cell (make-posn 0 1) false))
              (make-posn 0 1)
              'cat-won
              2))
 (overlay
  (board->image (list (make-cell (make-posn 0 1) false))
                2)
  (move-pinhole happy-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))
```

```
(check-expect
 (world->image
  (make-world (list (make-cell (make-posn 0 1) false))
              (make-posn 0 1)
              'cat-lost
              2))
 (overlay
  (board->image (list (make-cell (make-posn 0 1) false))
                2)
  (move-pinhole sad-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))

; ;;chop-whiskers:image->image
; ;;cropstheimagesothatanythingaboveortotheleftofthepinholeisgone
(define (chop-whiskers img)
  (shrink img
          0
          0
          (- (image-width img) (pinhole-x img) 1)
          (- (image-height img) (pinhole-y img) 1)))

(check-expect (chop-whiskers (rectangle 5 5 'solid 'black))
              (put-pinhole (rectangle 3 3 'solid 'black) 0 0))
(check-expect (chop-whiskers (rectangle 6 6 'solid 'black))
              (put-pinhole (rectangle 3 3 'solid 'black) 0 0))

(check-expect
 (pinhole-x
  (world->image
   (make-world
    (list (make-cell (make-posn 0 0) false)
          (make-cell (make-posn 0 1) false)
          (make-cell (make-posn 1 0) false))
    (make-posn 0 0)
    'playing
    2)))
 0)
(check-expect
 (pinhole-x
  (world->image
   (make-world
    (list (make-cell (make-posn 0 0) false)
          (make-cell (make-posn 0 1) false)
          (make-cell (make-posn 1 0) false))
    (make-posn 0 1)
```

```
    'playing
    2)))
 0)


; ;;board->image:boardnumber->image
(define (board->image cs world-size)
  (foldl (lambda (x y) (overlay y x))
         (nw:rectangle (world-width world-size)
                       (world-height world-size)
                       'solid
                       'white)
         (map cell->image cs)))

(check-expect (board->image (list (make-cell (make-posn 0 0) false)) 3)
              (overlay
               (nw:rectangle (world-width 3)
                             (world-height 3)
                             'solid
                             'white)
               (cell->image (make-cell (make-posn 0 0) false))))


; ;;cell->image:cell->image
(define (cell->image c)
  (local [(define x (cell-center-x (cell-p c)))
          (define y (cell-center-y (cell-p c)))]
    (move-pinhole
     (cond
       [(cell-blocked? c)
        (circle circle-radius 'solid 'black)]
       [else
        (circle circle-radius 'solid 'lightblue)])
     (- x)
     (- y))))

(check-expect (cell->image (make-cell (make-posn 0 0) false))
              (move-pinhole (circle circle-radius 'solid 'lightblue)
                            (- circle-radius)
                            (- circle-radius)))
(check-expect (cell->image (make-cell (make-posn 0 0) true))
              (move-pinhole (circle circle-radius 'solid 'black)
                            (- circle-radius)
                            (- circle-radius)))

; ;;world-width:number->number
```

```
; ;;computesthewidthofthedrawnworldintermsofitssize
(define (world-width board-size)
  (local [(define rightmost-posn
            (make-posn (- board-size 1) (- board-size 2)))]
    (+ (cell-center-x rightmost-posn) circle-radius)))

(check-expect (world-width 3) 150)

; ;;world-height:number->number
; ;;computestheheightofthedrawnworldintermsofitssize
(define (world-height board-size)
  (local [(define bottommost-posn
            (make-posn (- board-size 1) (- board-size 1)))]
    (+ (cell-center-y bottommost-posn) circle-radius)))
(check-expect (world-height 3) 116.208)


; ;;cell-center-x:posn->number
(define (cell-center-x p)
  (local [(define x (posn-x p))
          (define y (posn-y p))]
    (+ circle-radius
       (* x circle-spacing 2)
       (if (odd? y)
           circle-spacing
           0))))

(check-expect (cell-center-x (make-posn 0 0))
              circle-radius)
(check-expect (cell-center-x (make-posn 0 1))
              (+ circle-spacing circle-radius))
(check-expect (cell-center-x (make-posn 1 0))
              (+ (* 2 circle-spacing) circle-radius))
(check-expect (cell-center-x (make-posn 1 1))
              (+ (* 3 circle-spacing) circle-radius))

; ;;cell-center-y:posn->number
(define (cell-center-y p)
  (local [(define y (posn-y p))]
    (+ circle-radius
       (* y circle-spacing 2
          0.866))))


(check-expect (cell-center-y (make-posn 1 1))
              (+ circle-radius (* 2 circle-spacing 0.866)))
```

```
(check-expect (cell-center-y (make-posn 1 0))
              circle-radius)


; ;
; ;
; ;
; ;
; ;
; ;;;;;;
; ;;;;;
; ;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;
; ;;;;;;;;
; ;

; ;;adistance-mapis
; ;;(listofdist-cells)

; ;;adist-cellis
; ;;-(make-dist-cellposn(numberor'))
(define-struct dist-cell (p n))

; ;;build-table/fast:world->distance-map
(define (build-table/fast world)
  (local [(define board-size (world-size world))
          (define blocked (make-hash))
          (define ht (make-hash))
          (define (search p)
            (cond
              [(hash-ref blocked p)
               ']
              [(on-boundary? p board-size)
               ((lambda (a b) b)
                (hash-set! ht p 0)
                0)]
              [(not (boolean? (hash-ref ht p #f)))
               (hash-ref ht p)]
```

```
              [else
               ((lambda (a b c) c)
                (hash-set! ht p ')
                (hash-set!
                 ht
                 p
                 (add1/f (min-l (map search
                                     (adjacent p board-size)))))
                (hash-ref ht p))]))]
    ((lambda (a b c) c)
     (for-each (lambda (cell)
                 (hash-set! blocked
                            (cell-p cell)
                            (cell-blocked? cell)))
               (world-board world))
     (search (world-cat world))
     (hash-map ht make-dist-cell))))

; ;;build-table:world->distance-map
(define (build-table world)
  (build-distance (world-board world)
                  (world-cat world)
                  '()
                  '()
                  (world-size world)))

; ;;build-distance:boardposndistance-map(listofposn)number->distance-map
(define (build-distance board p t visited board-size)
  (cond
    [(cell-blocked? (lookup-board board p))
     (add-to-table p ' t)]
    [(on-boundary? p board-size)
     (add-to-table p 0 t)]
    [(in-table? t p)
     t]
    [(member p visited)
     (add-to-table p ' t)]
    [else
     (local [(define neighbors (adjacent p board-size))
             (define neighbors-t (build-distances
                                  board
                                  neighbors
                                  t
                                  (cons p visited)
                                  board-size))]
       (add-to-table p
```

```
                    (add1/f
                     (min-l
                      (map (lambda (neighbor)
                             (lookup-in-table neighbors-t neighbor))
                           neighbors)))
                    neighbors-t))]))

; ;;build-distances:board(listofposn)distance-map(listofposn)number
; ;;->distance-map
(define (build-distances board ps t visited board-size)
  (cond
    [(empty? ps) t]
    [else
     (build-distances board
                      (rest ps)
                      (build-distance board (first ps) t visited board-size)
                      visited
                      board-size)]))

(check-expect (build-distance (list (make-cell (make-posn 0 0) false))
                              (make-posn 0 0)
                              '()
                              '()
                              1)
              (list (make-dist-cell (make-posn 0 0) 0)))

(check-expect (build-distance (list (make-cell (make-posn 0 0) true))
                              (make-posn 0 0)
                              '()
                              '()
                              1)
              (list (make-dist-cell (make-posn 0 0) ')))

(check-expect (build-distance (list (make-cell (make-posn 0 1) false)
                                    (make-cell (make-posn 1 0) false)
                                    (make-cell (make-posn 1 1) false)
                                    (make-cell (make-posn 1 2) false)
                                    (make-cell (make-posn 2 0) false)
                                    (make-cell (make-posn 2 1) false)
                                    (make-cell (make-posn 2 2) false))
                              (make-posn 1 1)
                              '()
                              '()
                              3)
              (list (make-dist-cell (make-posn 1 0) 0)
                    (make-dist-cell (make-posn 2 0) 0)
```

```
                    (make-dist-cell (make-posn 0 1) 0)
                    (make-dist-cell (make-posn 2 1) 0)
                    (make-dist-cell (make-posn 1 2) 0)
                    (make-dist-cell (make-posn 2 2) 0)
                    (make-dist-cell (make-posn 1 1) 1)))

(check-expect (build-distance (list (make-cell (make-posn 0 1) true)
                                    (make-cell (make-posn 1 0) true)
                                    (make-cell (make-posn 1 1) false)
                                    (make-cell (make-posn 1 2) true)
                                    (make-cell (make-posn 2 0) true)
                                    (make-cell (make-posn 2 1) true)
                                    (make-cell (make-posn 2 2) true))
                              (make-posn 1 1)
                              '()
                              '()
                              3)
              (list (make-dist-cell (make-posn 1 0) ')
                    (make-dist-cell (make-posn 2 0) ')
                    (make-dist-cell (make-posn 0 1) ')
                    (make-dist-cell (make-posn 2 1) ')
                    (make-dist-cell (make-posn 1 2) ')
                    (make-dist-cell (make-posn 2 2) ')
                    (make-dist-cell (make-posn 1 1) ')))

(check-expect (build-distance
               (append-all
                (build-list
                 5
                 (lambda (i)
                   (build-list
                    5
                    (lambda (j)
                      (make-cell (make-posn i j) false))))))
               (make-posn 2 2)
               '()
               '()
               5)
              (list (make-dist-cell (make-posn 1 0) 0)
                    (make-dist-cell (make-posn 2 0) 0)
                    (make-dist-cell (make-posn 0 1) 0)
                    (make-dist-cell (make-posn 3 0) 0)
                    (make-dist-cell (make-posn 1 1) 1)
                    (make-dist-cell (make-posn 4 0) 0)
                    (make-dist-cell (make-posn 2 1) 1)
                    (make-dist-cell (make-posn 4 1) 0)
```

```
                    (make-dist-cell (make-posn 3 1) 1)
                    (make-dist-cell (make-posn 2 2) 2)
                    (make-dist-cell (make-posn 4 2) 0)
                    (make-dist-cell (make-posn 3 2) 1)
                    (make-dist-cell (make-posn 0 2) 0)
                    (make-dist-cell (make-posn 0 3) 0)
                    (make-dist-cell (make-posn 1 3) 1)
                    (make-dist-cell (make-posn 1 2) 1)
                    (make-dist-cell (make-posn 2 3) 1)
                    (make-dist-cell (make-posn 1 4) 0)
                    (make-dist-cell (make-posn 2 4) 0)
                    (make-dist-cell (make-posn 4 3) 0)
                    (make-dist-cell (make-posn 3 4) 0)
                    (make-dist-cell (make-posn 4 4) 0)
                    (make-dist-cell (make-posn 3 3) 1)))


; ;;lookup-board:boardposn->cell-or-false
(define (lookup-board board p)
  (cond
    [(empty? board) (error 'lookup-board "did not find posn")]
    [else
     (cond
       [(equal? (cell-p (first board)) p)
        (first board)]
       [else
        (lookup-board (rest board) p)])]))

(check-expect (lookup-board (list (make-cell (make-posn 2 2) false))
                            (make-posn 2 2))
              (make-cell (make-posn 2 2) false))
(check-error (lookup-board '() (make-posn 0 0))
             "lookup-board: did not find posn")

; ;;add-to-table:posn(numberor')distance-map->distance-map
(define (add-to-table p n t)
  (cond
    [(empty? t) (list (make-dist-cell p n))]
    [else
     (cond
       [(equal? p (dist-cell-p (first t)))
        (cons (make-dist-cell p (min/f (dist-cell-n (first t)) n))
              (rest t))]
       [else
        (cons (first t) (add-to-table p n (rest t)))])]))
```

```
(check-expect (add-to-table (make-posn 1 2) 3 '())
              (list (make-dist-cell (make-posn 1 2) 3)))
(check-expect (add-to-table (make-posn 1 2)
                            3
                            (list (make-dist-cell (make-posn 1 2) 4)))
              (list (make-dist-cell (make-posn 1 2) 3)))
(check-expect (add-to-table (make-posn 1 2)
                            3
                            (list (make-dist-cell (make-posn 1 2) 2)))
              (list (make-dist-cell (make-posn 1 2) 2)))
(check-expect (add-to-table (make-posn 1 2)
                            3
                            (list (make-dist-cell (make-posn 2 2) 2)))
              (list (make-dist-cell (make-posn 2 2) 2)
                    (make-dist-cell (make-posn 1 2) 3)))

; ;;in-table:distance-mapposn->boolean
(define (in-table? t p) (number? (lookup-in-table t p)))

(check-expect (in-table? empty (make-posn 1 2)) false)
(check-expect (in-table? (list (make-dist-cell (make-posn 1 2) 3))
                         (make-posn 1 2))
              true)
(check-expect (in-table? (list (make-dist-cell (make-posn 2 1) 3))
                         (make-posn 1 2))
              false)

; ;;lookup-in-table:distance-mapposn->numberor'
; ;;looksforthedistanceasrecordedinthetablet,
; ;;ifnotfoundreturnsadistanceof'
(define (lookup-in-table t p)
  (cond
    [(empty? t) ']
    [else (cond
            [(equal? p (dist-cell-p (first t)))
             (dist-cell-n (first t))]
            [else
             (lookup-in-table (rest t) p)])]))

(check-expect (lookup-in-table empty (make-posn 1 2)) ')
(check-expect (lookup-in-table (list (make-dist-cell (make-posn 1 2) 3))
                               (make-posn 1 2))
              3)
(check-expect (lookup-in-table (list (make-dist-cell (make-posn 2 1) 3))
                               (make-posn 1 2))
              ')
```

```
; ;;on-boundary?:posnnumber->boolean
(define (on-boundary? p board-size)
  (or (= (posn-x p) 0)
      (= (posn-y p) 0)
      (= (posn-x p) (- board-size 1))
      (= (posn-y p) (- board-size 1))))

(check-expect (on-boundary? (make-posn 0 1) 13) true)
(check-expect (on-boundary? (make-posn 1 0) 13) true)
(check-expect (on-boundary? (make-posn 12 1) 13) true)
(check-expect (on-boundary? (make-posn 1 12) 13) true)
(check-expect (on-boundary? (make-posn 1 1) 13) false)
(check-expect (on-boundary? (make-posn 10 10) 13) false)

; ;;adjacent:posnnumber->(listofposn)
(define (adjacent p board-size)
  (local [(define x (posn-x p))
          (define y (posn-y p))]
    (filter (lambda (x) (in-bounds? x board-size))
            (cond
              [(even? y)
               (list (make-posn (- x 1) (- y 1))
                     (make-posn x (- y 1))
                     (make-posn (- x 1) y)
                     (make-posn (+ x 1) y)
                     (make-posn (- x 1) (+ y 1))
                     (make-posn x (+ y 1)))]
              [else
               (list (make-posn x (- y 1))
                     (make-posn (+ x 1) (- y 1))
                     (make-posn (- x 1) y)
                     (make-posn (+ x 1) y)
                     (make-posn x (+ y 1))
                     (make-posn (+ x 1) (+ y 1)))])))))

(check-expect (adjacent (make-posn 1 1) 11)
              (list (make-posn 1 0)
                    (make-posn 2 0)
                    (make-posn 0 1)
                    (make-posn 2 1)
                    (make-posn 1 2)
                    (make-posn 2 2)))
(check-expect (adjacent (make-posn 2 2) 11)
              (list (make-posn 1 1)
                    (make-posn 2 1)
```

```
                    (make-posn 1 2)
                    (make-posn 3 2)
                    (make-posn 1 3)
                    (make-posn 2 3)))

; ;;in-bounds?:posnnumber->boolean
(define (in-bounds? p board-size)
  (and (<= 0 (posn-x p) (- board-size 1))
       (<= 0 (posn-y p) (- board-size 1))
       (not (equal? p (make-posn 0 0)))
       (not (equal? p (make-posn 0 (- board-size 1))))))
(check-expect (in-bounds? (make-posn 0 0) 11) false)
(check-expect (in-bounds? (make-posn 0 1) 11) true)
(check-expect (in-bounds? (make-posn 1 0) 11) true)
(check-expect (in-bounds? (make-posn 10 10) 11) true)
(check-expect (in-bounds? (make-posn 0 -1) 11) false)
(check-expect (in-bounds? (make-posn -1 0) 11) false)
(check-expect (in-bounds? (make-posn 0 11) 11) false)
(check-expect (in-bounds? (make-posn 11 0) 11) false)
(check-expect (in-bounds? (make-posn 10 0) 11) true)
(check-expect (in-bounds? (make-posn 0 10) 11) false)

; ;;min-l:(listofnumber-or-symbol)->number-or-symbol
(define (min-l ls) (foldr (lambda (x y) (min/f x y)) ' ls))
(check-expect (min-l (list)) ')
(check-expect (min-l (list 10 1 12)) 1)

; ;;<=/f:(numberor')(numberor')->boolean
(define (<=/f a b) (equal? a (min/f a b)))
(check-expect (<=/f 1 2) true)
(check-expect (<=/f 2 1) false)
(check-expect (<=/f ' 1) false)
(check-expect (<=/f 1 ') true)
(check-expect (<=/f ' ') true)

; ;;min/f:(numberor')(numberor')->(numberor')
(define (min/f x y)
  (cond
    [(equal? x ') y]
    [(equal? y ') x]
    [else (min x y)]))
(check-expect (min/f ' 1) 1)
(check-expect (min/f 1 ') 1)
(check-expect (min/f ' ') ')
(check-expect (min/f 1 2) 1)
```

```
; ;;add1/f:numberor'->numberor'
(define (add1/f n)
  (cond
    [(equal? n ') ']
    [else (add1 n)]))
(check-expect (add1/f 1) 2)
(check-expect (add1/f ') ')

; ;
; ;
; ;
; ;
; ;
; ;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;
; ;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;
; ;
; ;
; ;


(define (clack world x y evt)
  (cond
    [(and (equal? evt 'button-up)
          (equal? 'playing (world-state world))
          (point-in-circle? (world-board world) x y))
     (move-cat
      (make-world (add-obstacle (world-board world) x y)
                  (world-cat world)
                  (world-state world)
                  (world-size world)))]
    [else
     world]))

(check-expect (clack (make-world '() (make-posn 0 0) 'playing 1)
                     10
                     10
                     'button-down)
```

```
              (make-world '() (make-posn 0 0) 'playing 1))

(check-expect (clack (make-world '() (make-posn 0 0) 'playing 1)
                     0
                     0
                     'button-up)
              (make-world '() (make-posn 0 0) 'playing 1))

(check-expect (clack (make-world '() (make-posn 0 0) 'cat-lost 1)
                     10
                     10
                     'button-up)
              (make-world '() (make-posn 0 0) 'cat-lost 1))
(check-expect (clack
               (make-world
                (list (make-cell (make-posn 1 0) false)
                      (make-cell (make-posn 2 0) true)
                      (make-cell (make-posn 0 1) true)
                      (make-cell (make-posn 1 1) false)
                      (make-cell (make-posn 2 1) true)
                      (make-cell (make-posn 1 2) true)
                      (make-cell (make-posn 2 2) true))
                (make-posn 1 1)
                'playing
                3)
               (cell-center-x (make-posn 1 0))
               (cell-center-y (make-posn 1 0))
               'button-up)
              (make-world
               (list (make-cell (make-posn 1 0) true)
                     (make-cell (make-posn 2 0) true)
                     (make-cell (make-posn 0 1) true)
                     (make-cell (make-posn 1 1) false)
                     (make-cell (make-posn 2 1) true)
                     (make-cell (make-posn 1 2) true)
                     (make-cell (make-posn 2 2) true))
               (make-posn 1 1)
               'cat-lost
               3))

; ;;move-cat:world->world
(define (move-cat world)
  (local [(define cat-position (world-cat world))
          (define table (build-table/fast world))
          (define neighbors (adjacent cat-position (world-size world)))
          (define next-cat-positions
```

```
          (find-best-positions neighbors
                               (map (lambda (p) (lookup-in-table table p))
                                    neighbors)))
        (define next-cat-position
          (cond
            [(boolean? next-cat-positions) false]
            [else
             (list-ref next-cat-positions
                       (random (length next-cat-positions)))]))]
  (make-world (world-board world)
              (cond
                [(boolean? next-cat-position)
                 cat-position]
                [else next-cat-position])
              (cond
                [(boolean? next-cat-position)
                 'cat-lost]
                [(on-boundary? next-cat-position (world-size world))
                 'cat-won]
                [else 'playing])
              (world-size world))))


(check-expect
 (move-cat
  (make-world (list (make-cell (make-posn 1 0) false)
                    (make-cell (make-posn 2 0) false)
                    (make-cell (make-posn 3 0) false)
                    (make-cell (make-posn 4 0) false)

                    (make-cell (make-posn 0 1) false)
                    (make-cell (make-posn 1 1) true)
                    (make-cell (make-posn 2 1) true)
                    (make-cell (make-posn 3 1) false)
                    (make-cell (make-posn 4 1) false)

                    (make-cell (make-posn 0 2) false)
                    (make-cell (make-posn 1 2) true)
                    (make-cell (make-posn 2 2) false)
                    (make-cell (make-posn 3 2) true)
                    (make-cell (make-posn 4 2) false)

                    (make-cell (make-posn 0 3) false)
                    (make-cell (make-posn 1 3) true)
                    (make-cell (make-posn 2 3) false)
                    (make-cell (make-posn 3 3) false)
```

```
                    (make-cell (make-posn 4 3) false)

                    (make-cell (make-posn 1 4) false)
                    (make-cell (make-posn 2 4) false)
                    (make-cell (make-posn 3 4) false)
                    (make-cell (make-posn 4 4) false))
              (make-posn 2 2)
              'playing
              5))
 (make-world (list (make-cell (make-posn 1 0) false)
                   (make-cell (make-posn 2 0) false)
                   (make-cell (make-posn 3 0) false)
                   (make-cell (make-posn 4 0) false)

                   (make-cell (make-posn 0 1) false)
                   (make-cell (make-posn 1 1) true)
                   (make-cell (make-posn 2 1) true)
                   (make-cell (make-posn 3 1) false)
                   (make-cell (make-posn 4 1) false)

                   (make-cell (make-posn 0 2) false)
                   (make-cell (make-posn 1 2) true)
                   (make-cell (make-posn 2 2) false)
                   (make-cell (make-posn 3 2) true)
                   (make-cell (make-posn 4 2) false)

                   (make-cell (make-posn 0 3) false)
                   (make-cell (make-posn 1 3) true)
                   (make-cell (make-posn 2 3) false)
                   (make-cell (make-posn 3 3) false)
                   (make-cell (make-posn 4 3) false)

                   (make-cell (make-posn 1 4) false)
                   (make-cell (make-posn 2 4) false)
                   (make-cell (make-posn 3 4) false)
                   (make-cell (make-posn 4 4) false))
              (make-posn 2 3)
              'playing
              5))

; ;;find-best-positions:(nelistofposn)(nelistofnumberor')->(nelistofposn)orfalse
(define (find-best-positions posns scores)
  (local [(define best-score (foldl (lambda (x sofar)
                                      (if (<=/f x sofar)
                                          x
                                          sofar))
```

```
                                    (first scores)
                                    (rest scores)))]
    (cond
      [(symbol? best-score) false]
      [else
       (map
        second
        (filter (lambda (x) (equal? (first x) best-score))
                (map list scores posns)))])))
(check-expect (find-best-positions (list (make-posn 0 0)) (list 1))
              (list (make-posn 0 0)))
(check-expect (find-best-positions (list (make-posn 0 0)) (list '))
              false)
(check-expect (find-best-positions (list (make-posn 0 0)
                                         (make-posn 1 1))
                                   (list 1 2))
              (list (make-posn 0 0)))
(check-expect (find-best-positions (list (make-posn 0 0)
                                         (make-posn 1 1))
                                   (list 1 1))
              (list (make-posn 0 0)
                    (make-posn 1 1)))
(check-expect (find-best-positions (list (make-posn 0 0)
                                         (make-posn 1 1))
                                   (list ' 2))
              (list (make-posn 1 1)))
(check-expect (find-best-positions (list (make-posn 0 0)
                                         (make-posn 1 1))
                                   (list ' '))
              false)

; ;;add-obstacle:boardnumbernumber->board
(define (add-obstacle board x y)
  (cond
    [(empty? board) board]
    [else
     (local [(define cell (first board))
             (define cx (cell-center-x (cell-p cell)))
             (define cy (cell-center-y (cell-p cell)))]
       (cond
         [(and (<= (- cx circle-radius) x (+ cx circle-radius))
               (<= (- cy circle-radius) y (+ cy circle-radius)))
          (cons (make-cell (cell-p cell) true)
                (rest board))]
         [else
          (cons cell (add-obstacle (rest board) x y))]))]))
```

```
(check-expect (add-obstacle (list (make-cell (make-posn 0 0) false))
                            circle-spacing circle-spacing)
              (list (make-cell (make-posn 0 0) true)))
(check-expect (add-obstacle (list (make-cell (make-posn 0 0) false)) 100 100)
              (list (make-cell (make-posn 0 0) false)))
(check-expect (add-obstacle (list (make-cell (make-posn 0 0) false)
                                  (make-cell (make-posn 0 1) false))
                            circle-spacing circle-spacing)
              (list (make-cell (make-posn 0 0) true)
                    (make-cell (make-posn 0 1) false)))

; ;;point-in-circle?:boardnumbernumber->boolean
(define (point-in-circle? board x y)
  (cond
    [(empty? board) false]
    [else
     (local [(define cell (first board))
             (define center (+ (cell-center-x (cell-p cell))
                               (* (sqrt -1) (cell-center-y (cell-p cell)))))
             (define p (+ x (* (sqrt -1) y)))]
       (or (<= (magnitude (- center p)) circle-radius)
           (point-in-circle? (rest board) x y)))]))
(check-expect (point-in-circle? empty 0 0) false)
(check-expect (point-in-circle? (list (make-cell (make-posn 0 0) false))
                                (cell-center-x (make-posn 0 0))
                                (cell-center-y (make-posn 0 0)))
              true)
(check-expect (point-in-circle? (list (make-cell (make-posn 0 0) false))
                                0 0)
              false)




; ;
; ;
; ;
; ;
; ;
; ;;;;;
; ;;;;
; ;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;
```

```
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;
; ;
; ;
; ;


; ;;cat:symbol->image
(define (cat mode)
  (local [(define face-color
            (cond
              [(symbol=? mode 'sad) 'pink]
              [else 'lightgray]))

          (define left-ear (regular-polygon 3 8 'solid 'black (/ pi -3)))
          (define right-ear (regular-polygon 3 8 'solid 'black 0))
          (define ear-x-offset 14)
          (define ear-y-offset 9)

          (define eye (overlay (ellipse 12 8 'solid 'black)
                               (ellipse 6 4 'solid 'limegreen)))
          (define eye-x-offset 8)
          (define eye-y-offset 3)

          (define nose (regular-polygon 3 5 'solid 'black (/ pi 2)))

          (define mouth-happy
            (overlay (ellipse 8 8 'solid face-color)
                     (ellipse 8 8 'outline 'black)
                     (move-pinhole
                      (rectangle 10 5 'solid face-color)
                      0
                      4)))
          (define mouth-no-expression
            (overlay (ellipse 8 8 'solid face-color)
                     (ellipse 8 8 'outline face-color)
                     (rectangle 10 5 'solid face-color)))

          (define mouth
            (cond
              [(symbol=? mode 'happy) mouth-happy]
              [else mouth-no-expression]))
```

```
          (define mouth-x-offset 4)
          (define mouth-y-offset -5)]

    (add-line
     (add-line
      (add-line
       (add-line
        (add-line
         (add-line
          (overlay (move-pinhole left-ear (- ear-x-offset) ear-y-offset)
                   (move-pinhole right-ear (- ear-x-offset 1) ear-y-offset)
                   (ellipse 40 26 'solid 'black)
                   (ellipse 36 22 'solid face-color)
                   (move-pinhole mouth (- mouth-x-offset) mouth-y-offset)
                   (move-pinhole mouth mouth-x-offset mouth-y-offset)
                   (move-pinhole eye (- eye-x-offset) eye-y-offset)
                   (move-pinhole eye eye-x-offset eye-y-offset)
                   (move-pinhole nose -1 -4))
          6 4 30 12 'black)
         6 4 30 4 'black)
        6 4 30 -4 'black)
       -6 4 -30 12 'black)
      -6 4 -30 4 'black)
     -6 4 -30 -4 'black)))

(define happy-cat (cat 'happy))
(define sad-cat (cat 'sad))
(define thinking-cat (cat 'thinking))


; ;
; ;
; ;
; ;
; ;
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;
```

```
; ;;;;
; ;
; ;
; ;
; ;
; ;
; ;
; ;;;;;;;;
; ;;;;;;;;;
; ;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; ;;;;;;;;;;
; ;
; ;
; ;

; ;;append-all:(listof(listX))->(listofX)
(define (append-all ls)
  (foldr append empty ls))

(check-expect (append-all empty) empty)
(check-expect (append-all (list (list 1 2 3))) (list 1 2 3))
(check-expect (append-all (list (list 1) (list 2) (list 3)))
              (list 1 2 3))

; ;;add-n-random-blocked-cells:number(listofcell)number->(listofcell)
(define (add-n-random-blocked-cells n all-cells board-size)
  (cond
    [(zero? n) all-cells]
    [else
     (local [(define unblocked-cells
               (filter (lambda (x)
                         (let ([cat-cell? (and (= (posn-x (cell-p x)) (quotient board-size 2
                                               (= (posn-y (cell-p x)) (quotient board-size 2

                           (and (not (cell-blocked? x))
                                (not cat-cell?))))
                       all-cells))
             (define to-block (list-ref unblocked-cells
```

```
                                        (random (length unblocked-cells)))))]
       (add-n-random-blocked-cells
        (sub1 n)
        (map (lambda (c) (if (equal? to-block c)
                             (make-cell (cell-p c) true)
                             c))
             all-cells)
        board-size))]))

(check-expect (add-n-random-blocked-cells 0 (list (make-cell (make-posn 0 0) true)) 10)
              (list (make-cell (make-posn 0 0) true)))
(check-expect (add-n-random-blocked-cells 1 (list (make-cell (make-posn 0 0) false)) 10)
              (list (make-cell (make-posn 0 0) true)))

(define dummy
  (local
    [(define board-size 11)
     (define initial-board
       (add-n-random-blocked-cells
        6
        (filter
         (lambda (c)
           (not (and (= 0 (posn-x (cell-p c)))
                     (or (= 0 (posn-y (cell-p c)))
                         (= (- board-size 1)
                            (posn-y (cell-p c)))))))
         (append-all
          (build-list
           board-size
           (lambda (i)
             (build-list
              board-size
              (lambda (j)
                (make-cell (make-posn i j)
                           false)))))))
        board-size))
     (define initial-world
       (make-world initial-board
                   (make-posn (quotient board-size 2)
                              (quotient board-size 2))
                   'playing
                   board-size))]

    (and
     (big-bang (world-width board-size)
               (world-height board-size)
```

```
          1
          initial-world)
 (on-redraw world->image)
 (on-mouse-event clack))))
```

1.19  GCalc — Visual λ-Calculus

To play GCalc, run the `PLT Games` program. (Under Unix, it's called `plt-games`).

GCalc is a system for visually demonstrating the λ-Calculus (not really a game).

See the following for the principles:

`http://www.grame.fr/Research/GCalcul/Graphic_Calculus.html`

`ftp://ftp.grame.fr/pub/Documents/ICMC94LambdaCalc.pdf`

1.19.1 The Window Layout

The window is divided into three working areas, each made of cells. Cells hold cube objects, which can be dragged between cells (with a few exceptions that are listed below). The working areas are as follows:

- The right side is the storage area. This is used for saving objects – drag any cube to/from here. Note that cubes can be named for convenience.
- The left side is a panel of basic color cubes. These cells always contain a set of basic cubes that are used as the primitive building blocks all other values are made of. They cannot be overwritten. (Note that this includes a transparent cell.)
- The center part is the working panel. This is the main panel where new cubes are constructed. The center cell is similar to a storage cell, and the surrounding eight cells all perform some operation on this cell.

1.19.2 User Interaction

Right-click any cell except for the basic colors on the left panel, or hit escape or F10 for a menu of operations. The menu also includes the keyboard shortcuts for these operations.

1.19.3 Cube operations

There are six simple operations that are considered part of the simple graphic cube world. The operations correspond to six of the operation cells: a left-right composition is built using the left and the right cells, a top-bottom using the top and the bottom, and a front-back using the top-left and bottom-right. Dragging a cube to one of these cells will use the corresponding operator to combine it with the main cell's cube. Using a right mouse click on one of these cells can be used to cancel dragging an object to that cell, this is not really an undo feature: a right-click on the right cell always splits the main cube to two halves and throws the right side.

The colored cubes and the six basic operators make this simple domain, which is extended to form a λ-Calculus-like language by adding abstractions and applications. Right-clicking on a basic cube on the left panel creates an abstraction which is actually a lambda expression except that colors are used instead of syntactic variables. For example, if the main cell contains R|G (red-green on the left and right), then right-clicking the green cube on the left panel leaves us with λ G . R|G, which is visualized as R|G with a green circle. The last two operator cells are used for application of these abstractions: drag a function to the top-right to have it applied on the main cube, or to the bottom-left to have the main cube applied to it. As in the λ-Calculus, all abstractions have exactly one variable, use currying for multiple variables.

So far the result is a domain of colored cubes that can be used in the same way as the simple λ-Calculus. There is one last extension that goes one step further: function cubes can themselves be combined with other functions using the simple operations. This results in a form of "spatial functions" that behave differently in different parts of the cube according to the construction. For example, a left-right construction of two functions f|g operates on a given cube by applying f on its left part and g on its right part. You can use the preferences dialog to change a few aspects of the computation.

Use the Open Example menu entry to open a sample file that contains lots of useful objects: Church numerals, booleans, lists, Y-combinator, etc.

2 Implementing New Games

The game-starting console inspects the sub-collections of the "games" collection. If a sub-collection has an "info.ss" module (see setup/infotab), the following fields of the collection's "info.ss" file are used:

- game [required] : used as a module name in the sub-collection to load for the game; the module must provide a game@ unit (see scheme/unit) with no particular exports; the unit is invoked with no imports to start the game.
- name [defaults to the collection name] : used to label the game-starting button in the game console.
- game-icon [defaults to collection name with ".png"] : used as a path to a bitmap file that is used for the game button's label; this image should be 32 by 32 pixels and have a mask.
- game-set [defaults to "Other Games"] : a label used to group games that declare themselves to be in the same set.

To implement card games, see games/cards. Card games typically belong in the "Cards" game set.

3 Showing Scribbled Help

```
(require games/show-scribbling)
```

```
(show-scribbling mod-path section-tag) → (-> void?)
  mod-path : module-path?
  section-tag : string?
```

Returns a thunk for opening a Scribbled section in the user's HTML browser. The *mod-path* is the document's main source module, and *section-tag* specifies the section in the document.

4 Showing Text Help

```
(require games/show-help)
```

```
(show-help coll-path frame-title [verbatim?]) → (-> any)
  coll-path : (listof string?)
  frame-title : string?
  verbatim? : any/c = #f
```

Returns a thunk for showing a help window based on plain text. Multiple invocations of the thunk bring the same window to the foreground (until the user closes the window).

The help window displays `"doc.txt"` from the collection specified by *coll-path*.

The *frame-title* argument is used for the help window title.

If *verbatim?* is true, then `"doc.txt"` is displayed verbatim, otherwise it is formatted as follows:

- Any line of the form `**`....`**` is omitted.
- Any line that starts with `*` after whitespace is indented as a bullet point.
- Any line that contains only `-`s and is as long as the previous line causes the previous line to be formatted as a title.
- Other lines are paragraph-flowed to fit the window.