Games: Fun Examples

Version 4.2.1

July 30, 2009

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 L*i*, R*i*, T*i*, and B*i* 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

The goal of Chat Noir 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.

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

To get some insight into the cat's behavior, hold down the "h" key. It will show you the cells that are on the cat's shortest path to the edge, assuming that the cell underneath the mouse has been blocked, so you can experiment to see how the shortest paths change by moving your mouse around.

The game was inspired by the one at Game Design and has essentially the same rules. It also inspired the final project for the introductory programming course at the University of Chicago in the fall of 2008.

The remainder of this document explains the implementation of the Chat Noir game in a Literate Programming style.

1.18.1 Overview

Chat Noir is implemented using HtDP's universe library: `2htdp/universe` (although it only uses the "world" portions of that library). The program is divided up into six parts: the world data definition, an implementation of breadth-first search, code that handles drawing of the world, code that handles user input, and some code that builds an initial world and starts the game.

<*main*> **::=**

```
(require scheme/list scheme/math
         lang/private/imageeq
         (for-syntax scheme/base))
(require 2htdp/universe lang/posn scheme/contract)
<world>
<breadth-first-search>
<board->graph>
<cats-path>
<drawing-the-cat>
<drawing>
<input>
<tests>
<go>
```

Each section also comes with a series of test cases that are collected into the <tests> chunk at the end of the program.

<*tests*> **::=**

<test-infrastructure>
<world-tests>
<board->graph-tests>
<breadth-first-search-tests>
<cats-path-tests>
<drawing-tests>
<input-tests>

Each test case uses either `test`, a simple form that accepts two arguments and compares them with `equal?`, or `test/set` which accepts two lists and compares them as if they were sets.

In general, most of the test cases are left to the end of the document, organized in a series of chunks that match the functions being tested. Some of the test cases, however, provide nice illustrations of the behavior of the function and so are included in the function's description.

1.18.2 The World

The main data structure for Chat Noir is `world`. It comes with a few functions that construct empty worlds and test cases for them.

<***world***> **::=**

<cell-struct> <world-struct> <empty-world> <empty-board>
<blocked-cells> <block-cell>

<***world-tests***> **::=**

<empty-world-test> <empty-board-test> <blocked-cells-tests>

The main structure definition is the `world` struct.

<***world-struct***> **::=**

```
(define-struct/contract world ([board (listof cell?)]
                               [cat posn?]
                               [state (or/c 'playing
                                            'cat-won
                                            'cat-lost)]
                               [size (and/c natural-number/c
                                            odd?
                                            (>=/c 3))]
                               [mouse-posn (or/c #f posn?)]
                               [h-down? boolean?])
  #:transparent)
```

It consists of a structure with six fields:

- `board`: representing the state of the board as a list of `cell`s, one for each circle on the game.
- `cat`: a `posn` indicating the position of the cat (interpreting the `posn` in the way that they are interpreted for the `board` field),
- `state`: the state of the game, which can be one of
  - `'playing`, indicating that the game is still going; this is the initial state.
  - `'cat-won`, indicating that the game is over and the cat won, or
  - `'cat-lost`, indicating that the game is over and the cat lost.
- `size`: an odd natural number indicating the size of the board
- `mouse-posn`: a `posn` for the location of the mouse (or `#f` if the mouse is not in the window), and
- `h-down?`: a boolean indicating if the `h` key is being pushed down.

A `cell` is a structure with two fields:

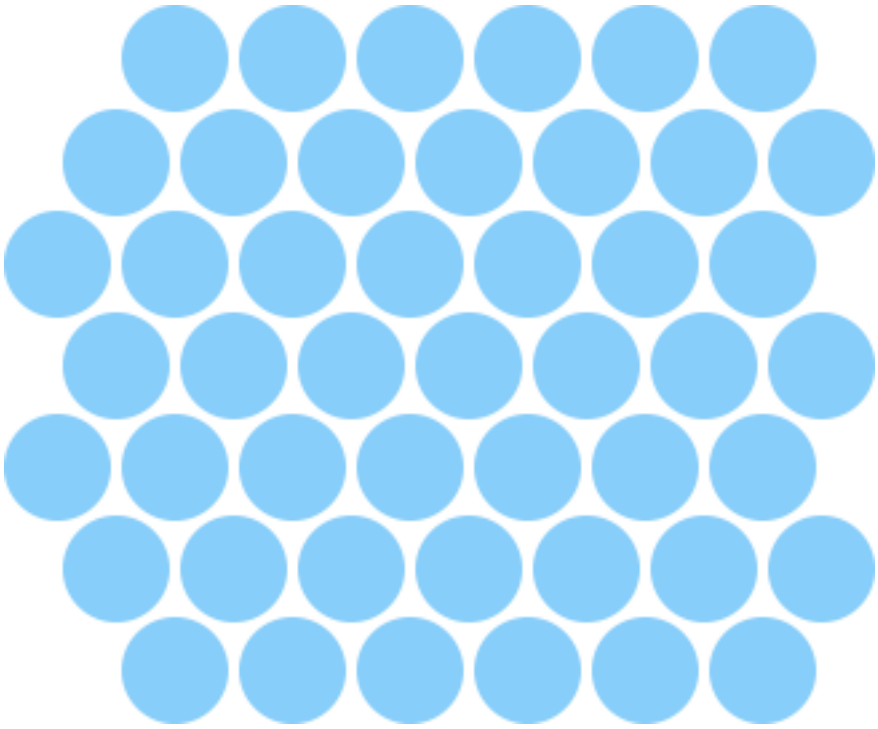
<*cell-struct*> **::=**

```
(define-struct/contract cell ([p posn?]
                              [blocked? boolean?])
  #:transparent)
```

The coordinates of the `posn` in the first field indicate a position on the hexagonal grid. This program reprsents the hexagon grid as a series of rows that are offset from each other by 1/2 the size of the each cell. The `y` field of the `posn` refers to the row of the cell, and the `x` coordinate the position in the row. This means that, for example, `(make-posn 1 0)` is centered above `(make-posn 1 0)` and `(make-posn 1 1)`.

The boolean in the `blocked?` field indicates if the cell has been clicked on, thus blocking the cat from stepping there.

The `empty-board` function builds a list of `cell`s that correspond to an empty board. For example, here's what an empty 7x7 board looks like, as a list of cells.



It contains 7 rows and, with the exception of the first and last rows, each row contains 7 cells. Notice how the even and odd rows are offset from each other by 1/2 of the size of the cell. The first and last row are missing their left-most cells because those cells are useless, from the perspective of the gameplay, Specifically, all of the neighbors of the missing cells are also on the boundary and thus the cat would win if it ever steps on one of those neighboring cells, ending the game.

The 3x3 board also has the same property that it consists of three rows, each with three cells, but where the first and last row are missing their left-most cells.

And here is how that board looks as a list of cells.

<*empty-board-test*> **::=**

```
(test (empty-board 3)
      (list
       (make-cell (make-posn 0 1) #f)
       (make-cell (make-posn 1 0) #f)
       (make-cell (make-posn 1 1) #f)
       (make-cell (make-posn 1 2) #f)
       (make-cell (make-posn 2 0) #f)
       (make-cell (make-posn 2 1) #f)
       (make-cell (make-posn 2 2) #f)))
```

The `empty-board` function consists of two (nested) calls to `build-list` that build a list of lists of cells, one for each pair of coordinates between `0` and `board-size`. Then, `append` flattens the nested lists and the `filter` expression removes the corners.

<*empty-board*> **::=**

```
(define/contract (empty-board board-size)
  (-> (and/c natural-number/c odd? (>=/c 3))
      (listof cell?))
  (filter
   (not-corner? board-size)
   (apply
    append
    (build-list
     board-size
     (lambda (i)
       (build-list
        board-size
        (lambda (j)
          (make-cell (make-posn i j)
                     #f))))))))

(define/contract ((not-corner? board-size) c)
  (-> (and/c natural-number/c odd? (>=/c 3))
      (-> cell?
          boolean?))
  (not (and (= 0 (posn-x (cell-p c)))
            (or (= 0 (posn-y (cell-p c)))
                (= (- board-size 1)
                   (posn-y (cell-p c)))))))
```

Building an empty world is simply a matter of building an empty board, finding the initial position of the cat and filling in all of the fields of the `world` struct. For example, this is the empty world of size `3`. It puts the cat at `(make-posn 1 1)`, sets the state to `'playing`,

records the size 3, and sets the current mouse position to #f and the state of the "h" key to #f.

<*empty-world-test*> **::=**

```
(test (empty-world 3)
      (make-world (empty-board 3)
                  (make-posn 1 1)
                  'playing
                  3
                  #f
                  #f))
```

The empty-world function generalizes the exmaple by computing the cats initial position as the center spot on the board.

<*empty-world*> **::=**

```
(define/contract (empty-world board-size)
  (-> (and/c natural-number/c odd? (>=/c 3))
      world?)
  (make-world (empty-board board-size)
              (make-posn (quotient board-size 2)
                         (quotient board-size 2))
              'playing
              board-size
              #f
              #f))
```

The add-n-random-blocked-cells function accepts a list of cells and returns a new list of cells where n of the unblocked cells in all-cells are now blocked.

If n is zero, of course, no more cells should be blocked, so the result is just all-cells. Otherwise, the function computes unblocked-cells, a list of all of the unblocked cells (except the cat's initial location), and then randomly picks a cell from it, calling block-cell to actually block that cell.

<*blocked-cells*> **::=**

```
(define/contract (add-n-random-blocked-cells n all-cells board-size)
  (-> natural-number/c (listof cell?) (and/c natural-number/c odd? (>=/c 3))
      (listof cell?))
  (cond
    [(zero? n) all-cells]
    [else
     (let* ([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)]
            [to-block (list-ref unblocked-cells
                                (random (length unblocked-cells)))])
       (add-n-random-blocked-cells
        (sub1 n)
        (block-cell (cell-p to-block) all-cells)
        board-size))]))
```

The `block-cell` function accepts a `posn` and a list of `cell` structs and updates the relevant cell, setting its `blocked?` field to `#t`.

<*block-cell*> **::=**

```
(define/contract (block-cell to-block board)
  (-> posn? (listof cell?) (listof cell?))
  (map (lambda (c) (if (equal? to-block (cell-p c))
                       (make-cell to-block #t)
                       c))
       board))
```

1.18.3 Breadth-first Search

The cat's move decision is based on a breadth-first search of a graph. The graph's nodes are the cells on the board plus a special node called `'boundary` that is adjacent to every cell on the boundary of the graph. In addition to the boundary edges, there are edges between each pair of adjacent cells, unless one of the cells is blocked, in which case it has no edges at all (even to the boundary).

This section describes the implementation of the breadth-first search, leaving details of how the graph connectivity is computed from the board to the next section.

<*breadth-first-search*> **::=**

<dist-cell-data-definition>
<lookup-in-table>
<build-bfs-table>

<*breadth-first-search-tests*> **::=**

<lookup-in-table-tests>
<build-bfs-table-tests>

The breadth-first function constructs a `distance-map`, which is a list of `dist-cell` structs:

<*dist-cell-data-definition*> **::=**

```
(define-struct/contract dist-cell ([p (or/c 'boundary posn?)]
                                   [n natural-number/c])
  #:transparent)
```

Each p field in the `dist-cell` is a position on the board and the n field is a natural number, indicating the distance of the shortest path from the node to some fixed point on the board.

The function `lookup-in-table` returns the distance from the fixed point to the given posn, returning `'∞` if the posn is not in the table.

<*lookup-in-table*> **::=**

```
(define/contract (lookup-in-table t p)
  (-> (listof dist-cell?) posn?
      (or/c '∞ natural-number/c))
  (cond
    [(empty? t) '∞]
    [else (cond
            [(equal? p (dist-cell-p (first t)))
             (dist-cell-n (first t))]
            [else
             (lookup-in-table (rest t) p)])]))
```

The `build-bfs-table` accepts a world and a cell (indicating the fixed point) and returns a distance map encoding the distance to that cell. For example, here is the distance map for the distance to the boundary.

<*build-bfs-table-tests*> **::=**

```
(test/set (build-bfs-table (empty-world 3)
                           'boundary)
          (list
           (make-dist-cell 'boundary 0)

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

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

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

The boundary is zero steps away; each of the cells that are on the boundary are one step away and the center is two steps away.

The core of the breadth-first search is this function, `bst`. It accepts a queue of the pending nodes to visit and a `dist-table` that records the same information as a `distance-map`, but in an immutable hash-table. The `dist-map` is an accumulator, recording the distances to all of the nodes that have already been visited in the graph, and is used here to speed up the compuation. The queue is represented as a list of vectors of length two. Each element in the queue contains a `posn`, or the symbol `'boundary` and that `posn`'s distance.

<*bfs*> **::=**

```
(define/contract (bfs queue dist-table)
  (-> (listof (vector/c (or/c 'boundary posn?) natural-number/c))
      hash?
      hash?)
  #:freevar neighbors/w (-> (or/c 'boundary posn?)
                            (listof (or/c 'boundary posn?)))
  (cond
    [(empty? queue) dist-table]
    [else
     (let* ([p (vector-ref (first queue) 0)]
            [dist (vector-ref (first queue) 1)])
       (cond
         [(hash-ref dist-table p #f)
          (bfs (rest queue) dist-table)]
         [else
          (bfs
           (append (rest queue)
                   (map (λ (p) (vector p (+ dist 1)))
                        (neighbors/w p)))
           (hash-set dist-table p dist))]))]))
```

If the `queue` is empty, then the accumulator contains bindings for all of the (reachable) nodes in the graph, so we just return it. If it isn't empty, then we extract the first element from the queue and name its consituents `p` and `dist`. Next we check to see if the node at the head of the queue is in `dist-table`. If it is, we just move on to the next element in the queue. If that node is not in the `dist-table`, then we add all of the neighbors to the queue, in the `append` expression, and update the `dist-table` with the distance to this node. Because we always add the new children to the end of the queue and always look at the front of the queue, we are guaranteed that the first time we see a node, it will be with the shortest distance.

The `build-bfs-table` function packages up `bfs` function. It accepts a `world` and an initial position and returns a `distance-table`.

<*build-bfs-table*> **::=**

```
(define/contract (build-bfs-table world init-point)
  (-> world? (or/c 'boundary posn?)
      (listof dist-cell?))
  (define neighbors/w (neighbors world))
  <bfs>

  (hash-map
   (bfs (list (vector init-point 0))
        (make-immutable-hash '()))
   make-dist-cell))
```

As you can see, the first thing it does is bind the free variable in `bfs` to the result of calling the `neighbors` function (defined in the chunk <neighbors>) and then it has the <bfs> chunk. In the body it calls the `bfs` function and then transforms the result, using `hash-map`, into a list of `cell`s.

1.18.4 Board to Graph

As far as the `build-bfs-table` function goes, all of the information specific to Chat Noir is encoded in the neighbors function. It accepts a world and returns a function that computes the neighbors of the boundary and of nodes. This section describes how it is implemented.

<***board-*>*graph*> ::=**

<neighbors>
<neighbors-blocked/boundary>
<adjacent>
<in-bounds?>
<on-boundary?>

<***board-*>*graph-tests*> ::=**

<on-boundary?-tests>
<in-bounds?-tests>
<adjacent-tests>
<neighbors-tests>

The neighbors functions accepts a `world` and then returns a function that computes the neighbors of a `posn` and of the `'boundary`.

For example, `(make-posn 1 0)` has four neighbors:

<***neighbors-tests*> ::=**

```
(test ((neighbors (empty-world 7)) (make-posn 1 0))
      (list 'boundary
            (make-posn 2 0)
            (make-posn 0 1)
            (make-posn 1 1)))
```

and `(make-posn 0 1)` has four neighbors:

<***neighbors-tests*> ::=**

```
(test ((neighbors (empty-world 7)) (make-posn 0 1))
      (list 'boundary
            (make-posn 1 0)
            (make-posn 1 1)
            (make-posn 0 2)
            (make-posn 1 2)))
```

as you can see in the earlier pictures of the 7x7 empty board. Also, there are 6 neighbors of the boundary in the 3x3 board:

<*neighbors-tests*> **::=**

```
(test ((neighbors (empty-world 3)) 'boundary)
      (list (make-posn 0 1)
            (make-posn 1 0)
            (make-posn 1 2)
            (make-posn 2 0)
            (make-posn 2 1)
            (make-posn 2 2)))
```

This is the neighbors function. After it accepts the `world`, it builds a list of the blocked cells in the world and a list of the cells that are on the boundary (and not blocked). Then it returns a function that is specialized to those values.

<*neighbors*> **::=**

```
(define/contract (neighbors w)
  (-> world?
      (-> (or/c 'boundary posn?)
          (listof (or/c 'boundary posn?))))
  (define blocked
    (map cell-p
         (filter (lambda (c)
                   (or (cell-blocked? c)
                       (equal? (cell-p c) (world-mouse-posn w))))
                 (world-board w))))
  (define boundary-cells
    (filter (lambda (p)
              (and (not (member p blocked))
                   (on-boundary? p (world-size w))))
            (map cell-p (world-board w))))
  (λ (p)
    (neighbors-blocked/boundary blocked
                                boundary-cells
                                (world-size w)
                                p)))
```

The neighbors-blocked/boundary function is given next. If p is blocked, it returns the empty list. If it is on the boundary, the function simply returns boundary-cells. Otherwise, neighbors-blocked/boundary calls adjacent to compute the posns that are adjacent to p, filtering out the blocked posns and binds that to adjacent-posns. It then filters out the posns that would be outside of the board. If those two lists are the same, then p is not on the boundary, so we just return in-bounds. If the lists are different, then we know that p must have been on the boundary, so we add 'boundary to the result list.

<*neighbors-blocked/boundary*> **::=**

```
(define/contract (neighbors-blocked/boundary blocked
                                             boundary-cells
                                             size
                                             p)
  (-> (listof posn?)
      (listof posn?)
      natural-number/c
      (or/c 'boundary posn?)
      (listof (or/c 'boundary posn?)))

  (cond
    [(member p blocked)
     '()]
    [(equal? p 'boundary)
     boundary-cells]
    [else
     (let* ([x (posn-x p)]
            [adjacent-posns
             (filter (λ (x) (not (member x blocked)))
                     (adjacent p))]
            [in-bounds
             (filter (λ (x) (in-bounds? x size))
                     adjacent-posns)])
       (cond
         [(equal? in-bounds adjacent-posns)
          in-bounds]
         [else
          (cons 'boundary in-bounds)]))]))
```

There are the three functions that build the basic graph structure from a board as used by neighbors.

The first function is adjacent. It consumes a posn and returns six posns that indicate what the neighbors are, without consideration of the size of the board (or the missing corner pieces).

For example, these are the posns that are adjacent to (make-posn 0 1); note that the first and the third are not on the board and do not show up in neighbors function example above.

<*adjacent-tests*> **::=**

```
(test (adjacent (make-posn 0 1))
      (list (make-posn 0 0)
            (make-posn 1 0)
            (make-posn -1 1)
            (make-posn 1 1)
            (make-posn 0 2)
            (make-posn 1 2)))
```

The adjacent function has two main cases; first when the y coordinate of the posn is even and second when it is odd. In each case, it is just a matter of looking at the board and calculating coordinate offsets.

<*adjacent*> **::=**

```
(define/contract (adjacent p)
  (-> posn?
      (and/c (listof posn?)
             (lambda (l) (= 6 (length l)))))
  (let ([x (posn-x p)]
        [y (posn-y p)])
    (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)))]))))
```

The on-boundary? function returns #t when the posn would be on the boundary of a board of size board-size. Note that this function does not have to special case the missing posns from the corners.

<*on-boundary?*> **::=**

```
(define/contract (on-boundary? p board-size)
  (-> posn? natural-number/c
      boolean?)
  (or (= (posn-x p) 0)
      (= (posn-y p) 0)
      (= (posn-x p) (- board-size 1))
      (= (posn-y p) (- board-size 1))))
```

The `in-bounds?` function returns `#t` when the `posn` is actually on the board, meaning that the coordinates of the `posn` are within the board's size, and that the `posn` is not one of the two corners that have been removed.

<*in-bounds?*> **::=**

```
(define/contract (in-bounds? p board-size)
  (-> posn? natural-number/c
      boolean?)
  (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))))))
```

1.18.5 The Cat's Path

Once we have a breadth-first search all sorted out, we can use it to build a function that determines where the shortest paths from the cat's current position to the boundary are.

<*cats-path*> **::=**

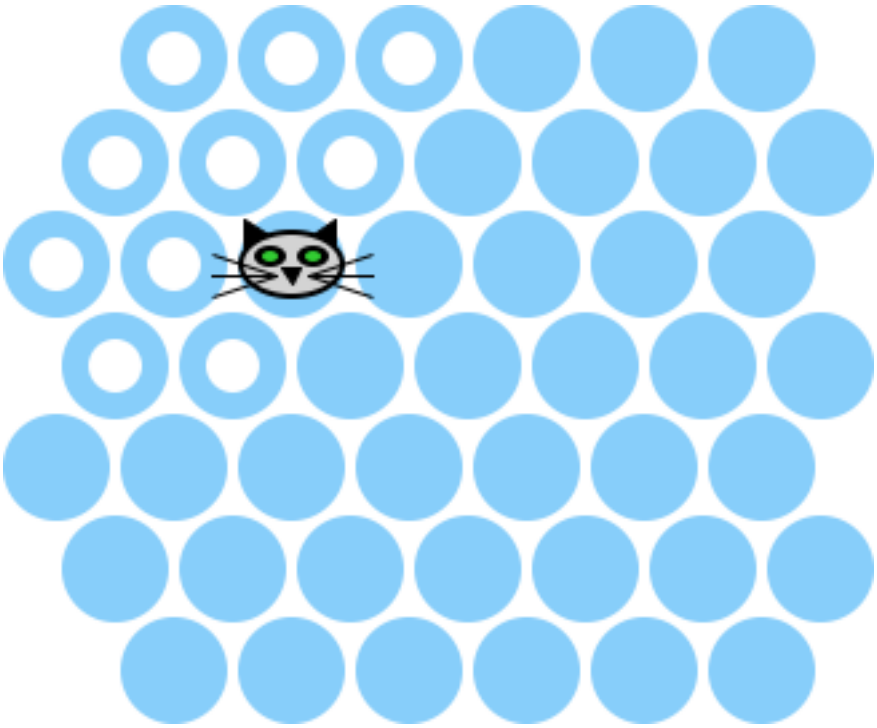
<on-cats-path?>
<+/f>

<*cats-path-tests*> **::=**

<on-cats-path?-tests>
<+/f-tests>

The function `on-cats-path?` accepts a world and returns a predicate on the `posn`s in the world. The predicate indicates if the given `posn` is on the shortest path.

For example, in a world of size `7` with the cat at `(make-posn 2 2)`, the circles with white centers are on the shortest path to the boundary:



So we can formulate two test cases using this world, one in the white circles and one not:

<on-cats-path?-tests> **::=**

```
(let ([on-the-path?
       (on-cats-path? (make-world (empty-board 7)
                                  (make-posn 2 2)
                                  'playing
                                  7
                                  #f
                                  #t))])
  (test (on-the-path? (make-posn 1 0))
        #t)
  (test (on-the-path? (make-posn 4 4))
        #f))
```

The computation of the shortest path to the boundary proceeds by computing two distance maps; the distance map to the boundary and the distance map to the cat. Then, a node is on one of the shortest paths if the distance to the cat plus the distance to the boundary is equal to the distance from the cat to the boundary.

The code is essentially that, plus two other special cases. Specifically if the "h" key is not pressed down, then we just consider no cells to be on that shortest path. And if the distance

to the cat is `'∞`, then again no nodes are on the path. The second situation happens when the cat is completely boxed in and has lost the game.

<*on-cats-path?*> **::=**

```
(define/contract (on-cats-path? w)
  (-> world? (-> posn? boolean?))
  (cond
    [(world-h-down? w)
     (let ()
       (define edge-distance-map (build-bfs-table w 'boundary))
       (define cat-distance-map (build-bfs-table w (world-cat w)))
       (define cat-distance
         (lookup-in-table edge-distance-map (world-cat w)))
       (cond
         [(equal? cat-distance '∞)
          (lambda (p) #f)]
         [else
          (lambda (p)
            (equal? (+/f (lookup-in-table cat-distance-map p)
                         (lookup-in-table edge-distance-map p))
                    cat-distance))]))]
    [else
     (lambda (p) #f)]))
```

Finally, the helper function `+/f` is just like `+`, except that it returns `'∞` if either argument is `'∞`.

<*+/f*> **::=**

```
(define (+/f x y)
  (cond
    [(or (equal? x '∞) (equal? y '∞))
     '∞]
    [else
     (+ x y)]))
```

1.18.6 Drawing the Cat

This code is three large, similar constants, bundled up into the `cat` function. The `thinking-cat` is the one that is visible when the game is being played. It differs from the others in that it does not have a mouth. The `mad-cat` is the one that you see when the cat loses. It differs from the others in that its pinks turn pink. Finally, the `happy-cat` shows up when the cat wins and it is just like the `thinking-cat` except it has a smile.

<*drawing-the-cat*> **::=**

```
(define/contract (cat mode)
  (-> (or/c 'mad 'happy 'thinking) image?)
  (define face-width 36)
  (define face-height 22)

  (define face-color
    (cond
      [(eq? mode 'mad) '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
      [(eq? mode 'happy) mouth-happy]
      [else mouth-no-expression]))
  (define mouth-x-offset 4)
  (define mouth-y-offset -5)

  (define (whiskers img)
    (add-line
     (add-line
      (add-line
       (add-line
        (add-line
         (add-line
          img
          6 4 30 12 'black)
         6 4 30 4 'black)
        6 4 30 -4 'black)
       -6 4 -30 12 'black)
```

1.18.7 Drawing the World

<*drawing*> **::=**

<constants>
<render-world>
<chop-whiskers>
<render-board>
<render-cell>
<world-width>
<world-height>
<cell-center-x>
<cell-center-y>

<*drawing-tests*> **::=**

<cell-center-x-tests>
<cell-center-y-tests>
<world-size-tests>
<render-cell-tests>
<render-board-tests>
<chop-whiskers-tests>
<render-world-tests>

There are a number of constants that are given names to make the code more readable.

These first two constants give the radius of the circles that are drawn on the board, plus the radius of an invisible circle that, if they were drawn on top of the circles, would touch each other. Accordingly, `circle-spacing` is used when computing the positions of the circles, but the circles are drawn using `circle-radius`.

<*constants*> **::=**

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

The other four constants specify the colors of the circles.

<*constants*> **::=**

```
(define normal-color 'lightskyblue)
(define on-shortest-path-color 'white)
(define blocked-color 'black)
(define under-mouse-color 'black)
```

The main function for drawing a world is `render-world`. It is a fairly straightforward composition of helper functions. First, it builds the image of a board, and then puts the cat

on it. Lastly, since the whiskers of the cat might now hang off of the edge of the board (if the cat is on a leftmost or rightmost cell), it trims them. This ensures that the image is always the same size and that the pinhole is always in the upper-left corner of the window.

<*render-world*> **::=**

```
(define/contract (render-world w)
  (-> world? image?)
  (chop-whiskers
   (overlay/xy (render-board (world-board w)
                             (world-size w)
                             (on-cats-path? w)
                             (world-mouse-posn w))
               (cell-center-x (world-cat w))
               (cell-center-y (world-cat w))
               (cond
                 [(equal? (world-state w) 'cat-won) happy-cat]
                 [(equal? (world-state w) 'cat-lost) mad-cat]
                 [else thinking-cat]))))
```

Trimming the cat's whiskers amounts to removing any extra space in the image that appears to the left or above the pinhole. For example, the `rectangle` function returns an image with a pinhole in the middle. So trimming 5x5 rectangle results in a 3x3 rectangle with the pinhole at (0,0).

<*chop-whiskers-tests*> **::=**

```
(test (chop-whiskers (rectangle 5 5 'solid 'black))
      (put-pinhole (rectangle 3 3 'solid 'black) 0 0))
```

The function uses `shrink` to remove all of the material above and to the left of the pinhole.

<*chop-whiskers*> **::=**

```
(define/contract (chop-whiskers img)
  (-> image? image?)
  (shrink img
          0
          0
          (- (image-width img) (pinhole-x img) 1)
          (- (image-height img) (pinhole-y img) 1)))
```

The `render-board` function uses `for/fold` to iterate over all of the `cells` in `cs`. It starts with an empty rectangle and, one by one, puts the cells on `image`.

<*render-board*> **::=**

```
(define/contract (render-board cs world-size on-cat-path? mouse)
  (-> (listof cell?)
      natural-number/c
      (-> posn? boolean?)
      (or/c #f posn?)
      image?)
  (for/fold ([image (nw:rectangle (world-width world-size)
                                  (world-height world-size)
                                  'solid
                                  'white)])
            ([c cs])
    (overlay image
             (render-cell c
                          (on-cat-path? (cell-p c))
                          (and (posn? mouse)
                               (equal? mouse (cell-p c)))))))
```

The `render-cell` function accepts a `cell`, a boolean indicating if the cell is on the shortest path between the cat and the boundary, and a second boolean indicating if the cell is underneath the mouse. It returns an image of the cell, with the pinhole placed in such a way that overlaying the image on an empty image with pinhole in the upper-left corner results in the cell being placed in the right place.

<render-cell> **::=**

```
(define/contract (render-cell c on-short-path? under-mouse?)
  (-> cell? boolean? boolean? image?)
  (let ([x (cell-center-x (cell-p c))]
        [y (cell-center-y (cell-p c))]
        [main-circle
         (cond
           [(cell-blocked? c)
            (circle circle-radius 'solid blocked-color)]
           [else
            (circle circle-radius 'solid normal-color)])])
    (move-pinhole
     (cond
       [under-mouse?
        (overlay main-circle
                 (circle (quotient circle-radius 2) 'solid under-mouse-color))]
       [on-short-path?
        (overlay main-circle
                 (circle (quotient circle-radius 2) 'solid
                         on-shortest-path-color))]
       [else
        main-circle])
     (- x)
     (- y))))
```

The `world-width` function computes the width of the rendered world, given the world's size by finding the center of the rightmost posn, and then adding an additional radius.

<*world-width*> **::=**

```
(define/contract (world-width board-size)
  (-> natural-number/c number?)
  (let ([rightmost-posn
         (make-posn (- board-size 1) (- board-size 2))])
    (+ (cell-center-x rightmost-posn) circle-radius)))
```

Similarly, the `world-height` function computest the height of the rendered world, given the world's size.

<*world-height*> **::=**

```
(define/contract (world-height board-size)
  (-> natural-number/c number?)
  (let ([bottommost-posn
         (make-posn (- board-size 1) (- board-size 1))])
    (ceiling (+ (cell-center-y bottommost-posn)
                circle-radius))))
```

The `cell-center-x` function returns the `x` coordinate of the center of the cell specified by `p`.

For example, the first cell in the third row (counting from `0`) is flush with the edge of the screen, so its center is just the radius of the circle that is drawn.

<*cell-center-x-tests*> **::=**

```
(test (cell-center-x (make-posn 0 2))
      circle-radius)
```

The first cell in the second row, in contrast is offset from the third row by `circle-spacing`.

<*cell-center-x-tests*> **::=**

```
(test (cell-center-x (make-posn 0 1))
      (+ circle-spacing circle-radius))
```

The definition of `cell-center-x` multiplies the `x` coordinate of `p` by twice `circle-spacing` and then adds `circle-radius` to move over for the first circle. In addition if the `y` coordinate is odd, then it adds `circle-spacing`, shifting the entire line over.

<*cell-center-x*> **::=**

```
(define/contract (cell-center-x p)
  (-> posn? number?)
  (let ([x (posn-x p)]
        [y (posn-y p)])
    (+ circle-radius
       (* x circle-spacing 2)
       (if (odd? y)
           circle-spacing
           0))))
```

The `cell-center-y` function computes the `y` coordinate of a cell's location on the screen. For example, the `y` coordinate of the first row is the radius of a circle, ensuring that the first row is flush against the top of the screen.

<*cell-center-y-tests*> **::=**

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

Because the grid is hexagonal, the `y` coordinates of the rows do not have the same spacing as the `x` coordinates. In particular, they are off by `sin(pi/3)`. We approximate that by `433/500` in order to keep the computations and test cases simple and using exact numbers. A more precise approximation would be `0.8660254037844386`, but it is not necessary at the screen resolution.

<*cell-center-y*> **::=**

```
(define/contract (cell-center-y p)
  (-> posn? number?)
  (+ circle-radius
     (* (posn-y p)
        circle-spacing 2
        433/500)))
```

1.18.8 Handling Input

Input handling consists of handling two different kinds of events: key events, and mouse events, plus various helper functions.

<*input*> **::=**

<change>
<clack>
<update-world-posn>
<player-moved?>
<block-cell/world>
<move-cat>
<find-best-positions>
<lt/f>
<circle-at-point>
<point-in-this-circle?>

<*input-tests*> **::=**

<change-tests>
<point-in-this-circle?-tests>
<circle-at-point-tests>
<lt/f-tests>
<find-best-positions-tests>
<move-cat-tests>
<update-world-posn-tests>
<clack-tests>

The `change` function handles keyboard input and merely updates the `h-down?` field based on the state of the key event.

<*change*> **::=**

```
(define (change w ke)
  (make-world (world-board w)
              (world-cat w)
              (world-state w)
              (world-size w)
              (world-mouse-posn w)
              (key=? ke "h")))
```

The `clack` function handles mouse input. It has three tasks and each corresponds to a helper function:

- block the clicked cell (`block-cell/world`),
- move the cat (`move-cat`), and
- update the black dot as the mouse moves around (`update-world-posn`).

The helper functions are combined in the body of `clack`, first checking to see if the mouse event corresponds to a player's move (via the `player-moved?` function.

<*clack*> **::=**

```
(define/contract (clack world x y evt)
  (-> world? integer? integer? any/c
      world?)
  (let ([moved-world
         (cond
           [(player-moved? world x y evt)
            =>
            (λ (circle)
              (move-cat
               (block-cell/world circle world)))]
           [else world])])
    (update-world-posn
     moved-world
     (and (eq? (world-state moved-world) 'playing)
          (not (equal? evt "leave"))
          (make-posn x y)))))
```

The `player-moved?` predicate returns a `posn` indicating where the player chose to move when the mouse event corresponds to a player move, and returns `#f`. It first checks to see if the mouse event is a button up event and that the game is not over, and then it just calls `circle-at-point`.

<*player-moved?*> **::=**

```
(define/contract (player-moved? world x y evt)
  (-> world? integer? integer? any/c
      (or/c posn? #f))
  (and (equal? evt "button-up")
       (equal? 'playing (world-state world))
       (circle-at-point (world-board world) x y)))
```

The `circle-at-point` function returns a `posn` when the coordinate (`x`,`y`) is inside a circle on the given board. Instead of computing the nearest circle to the coordinates, it simply iterates over the cells on the board and returns the `posn` of the matching cell.

<*circle-at-point*> **::=**

```
(define/contract (circle-at-point board x y)
  (-> (listof cell?) real? real?
      (or/c posn? #f))
  (ormap (λ (cell)
           (and (point-in-this-circle? (cell-p cell) x y)
                (cell-p cell)))
         board))
```

The `point-in-this-circle?` function returns `#t` when the point (`x`,`y`) on the screen falls within the circle located at the `posn` `p`.

This is precise about checking the circles. For example, a point that is (14,14) away from the center of a circle is still in the circle:

<*point-in-this-circle?-tests*> **::=**

```
(test (point-in-this-circle?
       (make-posn 1 0)
       (+ (cell-center-x (make-posn 1 0)) 14)
       (+ (cell-center-y (make-posn 1 0)) 14))
      #t)
```

but one that is (15,15) away is no longer in the circle, since it crosses the boundary away from a circle of radius 20 at that point.

<*point-in-this-circle?-tests*> **::=**

```
(test (point-in-this-circle?
       (make-posn 1 0)
       (+ (cell-center-x (make-posn 1 0)) 15)
       (+ (cell-center-y (make-posn 1 0)) 15))
      #f)
```

The implementation of `point-in-this-circle?` uses complex numbers to represent both points on the screen and directional vectors. In particular, the variable `center` is a complex

number whose real part is the x coordinate of the center of the cell at p, and its imaginary part is y coordinate. Similarly, mp is bound to a complex number corresponding to the position of the mouse, at (x, y). Then, the function computes the vector between the two points by subtracting the complex numbers from each other and extracting the magnitude from that vector.

<*point-in-this-circle?*> **::=**

```
(define/contract (point-in-this-circle? p x y)
  (-> posn? real? real? boolean?)
  (let ([center (+ (cell-center-x p)
                   (* (sqrt -1)
                      (cell-center-y p)))]
        [mp (+ x (* (sqrt -1) y))])
    (<= (magnitude (- center mp))
        circle-radius)))
```

In the event that player-moved? returns a posn, the clack function blocks the clicked on cell using block-cell/world, which simply calls block-cell.

<*block-cell/world*> **::=**

```
(define/contract (block-cell/world to-block w)
  (-> posn? world? world?)
  (make-world (block-cell to-block (world-board w))
              (world-cat w)
              (world-state w)
              (world-size w)
              (world-mouse-posn w)
              (world-h-down? w)))
```

The move-cat function uses calls build-bfs-table to find the shortest distance from all of the cells to the boundary, and then uses find-best-positions to compute the list of neighbors of the cat that have the shortest distance to the boundary. If that list is empty, then next-cat-position is #f, and otherwise, it is a random element from that list.

<*move-cat*> **::=**

```
(define/contract (move-cat world)
  (-> world? world?)
  (let* ([cat-position (world-cat world)]
         [table (build-bfs-table world 'boundary)]
         [neighbors (adjacent cat-position)]
         [next-cat-positions
          (find-best-positions neighbors
                               (map (lambda (p) (lookup-in-table table p))
                                    neighbors))]
         [next-cat-position
          (cond
            [(boolean? next-cat-positions) #f]
            [else
             (list-ref next-cat-positions
                       (random (length next-cat-positions)))])])
    <moved-cat-world>))
```

Once `next-cat-position` has been computed, it is used to update the `cat` and `state` fields of the world, recording the cat's new position and whether or not the cat won.

<***moved-cat-world***> **::=**

```
(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)
            (world-mouse-posn world)
            (world-h-down? world))
```

The `find-best-positions` function accepts two parallel lists, one of `posn`s, and one of scores for those `posn`s, and it returns either a non-empty list of `posn`s that have tied for the best score, or it returns `#f`, if the best score is `'`∞.

<***find-best-positions***> **::=**

```
(define/contract (find-best-positions posns scores)
  (-> (cons/c posn? (listof posn?))
      (cons/c (or/c number? '∞) (listof (or/c number? '∞)))
      (or/c (cons/c posn? (listof posn?)) #f))
  (let ([best-score
         (foldl (lambda (x sofar)
                  (if (<=/f x sofar)
                      x
                      sofar))
                (first scores)
                (rest scores))])
    (cond
      [(symbol? best-score) #f]
      [else
       (map
        second
        (filter (lambda (x) (equal? (first x) best-score))
                (map list scores posns)))])))
```

This is a helper function that behaves like `<=`, but is extended to deal properly with `'∞`.

<*lt/f*> **::=**

```
(define/contract (<=/f a b)
  (-> (or/c number? '∞)
      (or/c number? '∞)
      boolean?)
  (cond
    [(equal? b '∞) #t]
    [(equal? a '∞) #f]
    [else (<= a b)]))
```

Finally, to complete the mouse event handling, the `update-world-posn` function is called from `clack`. It updates the `mouse-down` field of the `world`. If the `p` argument is a `posn`, it corresponds to the location of the mouse, in graphical coordinates. So, the function converts it to a cell position on the board and uses that. Otherwise, when `p` is `#f`, the `mouse-down` field is just updated to `#f`.

<*update-world-posn*> **::=**

```
(define/contract (update-world-posn w p)
  (-> world? (or/c #f posn?)
      world?)
  (cond
    [(posn? p)
     (let ([mouse-spot
            (circle-at-point (world-board w)
                             (posn-x p)
                             (posn-y p))])
       (make-world (world-board w)
                   (world-cat w)
                   (world-state w)
                   (world-size w)
                   (cond
                     [(equal? mouse-spot (world-cat w))
                      #f]
                     [else
                      mouse-spot])
                   (world-h-down? w)))]
    [else
     (make-world (world-board w)
                 (world-cat w)
                 (world-state w)
                 (world-size w)
                 #f
                 (world-h-down? w))]))
```

1.18.9 Tests

This section consists of some infrastructure for maintaining tests, plus a pile of additional tests for the other functions in this document.

The `test` and `test/set` macros package up their arguments into thunks and then simply call `test/proc`, supplying information about the source location of the test case. The `test/proc` function runs the tests and reports the results.

<test-infrastructure> **::=**

```
(define-syntax (test stx)
  (syntax-case stx ()
    [(_ actual expected)
     (with-syntax ([line (syntax-line stx)]
                   [pos (syntax-position stx)])
       #'(test/proc (λ () actual)
                    (λ () expected)
                    equal?
                    line
                    'actual))]))

(define-syntax (test/set stx)
  (syntax-case stx ()
    [(_ actual expected)
     (with-syntax ([line (syntax-line stx)]
                   [pos (syntax-position stx)])
       #'(test/proc (λ () actual)
                    (λ () expected)
                    (λ (x y) (same-sets? x y))
                    line
                    'actual))]))

(define test-count 0)

(define (test/proc actual-thunk expected-thunk cmp line sexp)
  (set! test-count (+ test-count 1))
  (let ([actual (actual-thunk)]
        [expected (expected-thunk)])
    (unless (cmp actual expected)
      (error 'check-expect "test #~a~a\n  ~s\n  ~s\n"
             test-count
             (if line
                 (format " on line ~a failed:" line)
                 (format " failed: ~s" sexp))
             actual
             expected))))


(define (same-sets? l1 l2)
  (and (andmap (lambda (e1) (member e1 l2)) l1)
       (andmap (lambda (e2) (member e2 l1)) l2)
       #t))

(test (same-sets? (list) (list)) #t)
(test (same-sets? (list) (list 1)) #f)
(test (same-sets? (list 1) (list)) #f)
(test (same-sets? (list 1 2) (list 2 1)) #t)
```

<*lookup-in-table-tests*> **::=**

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

<*build-bfs-table-tests*> **::=**

```
(test/set (build-bfs-table
           (make-world (empty-board 3) (make-posn 1 1)
                       'playing 3 (make-posn 0 0) #f)
           (make-posn 1 1))
          (list
           (make-dist-cell 'boundary 2)

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

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

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

(test/set (build-bfs-table
           (make-world
            (list
             (make-cell (make-posn 0 1) #t)
             (make-cell (make-posn 1 0) #t)
             (make-cell (make-posn 1 1) #f)
             (make-cell (make-posn 1 2) #t)
             (make-cell (make-posn 2 0) #t)
             (make-cell (make-posn 2 1) #t)
             (make-cell (make-posn 2 2) #t))
            (make-posn 1 1)
            'playing
            3
            (make-posn 0 0)
            #f)
           'boundary)
          (list
           (make-dist-cell 'boundary 0)))

(test/set (build-bfs-table
           (make-world (empty-board 5)
                       (make-posn 2 2)
                       'playing
                       5
                       (make-posn 0 0)
                       #f)
           'boundary)
          (list
           (make-dist-cell 'boundary 0)

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

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

<*neighbors-tests*> **::=**

```
(test ((neighbors (empty-world 11))  (make-posn 1 1))
      (adjacent (make-posn 1 1)))
(test ((neighbors (empty-world 11)) (make-posn 2 2))
      (adjacent (make-posn 2 2)))
(test ((neighbors (empty-world 3)) 'boundary)
      (list (make-posn 0 1)
            (make-posn 1 0)
            (make-posn 1 2)
            (make-posn 2 0)
            (make-posn 2 1)
            (make-posn 2 2)))
(test ((neighbors (make-world (list
                               (make-cell (make-posn 0 1) #f)
                               (make-cell (make-posn 1 0) #f)
                               (make-cell (make-posn 1 1) #t)
                               (make-cell (make-posn 1 2) #f)
                               (make-cell (make-posn 2 0) #f)
                               (make-cell (make-posn 2 1) #f)
                               (make-cell (make-posn 2 2) #f))
                              (make-posn 1 1)
                              'playing
                              3
                              (make-posn 0 0)
                              #f))
       (make-posn 1 1))
      '())
(test ((neighbors (make-world (list
                               (make-cell (make-posn 0 1) #f)
                               (make-cell (make-posn 1 0) #f)
                               (make-cell (make-posn 1 1) #t)
                               (make-cell (make-posn 1 2) #f)
                               (make-cell (make-posn 2 0) #f)
                               (make-cell (make-posn 2 1) #f)
                               (make-cell (make-posn 2 2) #f))
                              (make-posn 1 1)
                              'playing
                              3
                              (make-posn 0 0)
                              #f))
       (make-posn 1 0))
      (list 'boundary (make-posn 2 0) (make-posn 0 1)))
```

<*adjacent-tests*> **::=**

```
  (test (adjacent (make-posn 1 1))
        (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)))
  (test (adjacent (make-posn 2 2))
        (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)))
```

<*on-boundary?-tests*> **::=**

```
  (test (on-boundary? (make-posn 0 1) 13) #t)
  (test (on-boundary? (make-posn 1 0) 13) #t)
  (test (on-boundary? (make-posn 12 1) 13) #t)
  (test (on-boundary? (make-posn 1 12) 13) #t)
  (test (on-boundary? (make-posn 1 1) 13) #f)
  (test (on-boundary? (make-posn 10 10) 13) #f)
```

<*in-bounds?-tests*> **::=**

```
  (test (in-bounds? (make-posn 0 0) 11) #f)
  (test (in-bounds? (make-posn 0 1) 11) #t)
  (test (in-bounds? (make-posn 1 0) 11) #t)
  (test (in-bounds? (make-posn 10 10) 11) #t)
  (test (in-bounds? (make-posn 0 -1) 11) #f)
  (test (in-bounds? (make-posn -1 0) 11) #f)
  (test (in-bounds? (make-posn 0 11) 11) #f)
  (test (in-bounds? (make-posn 11 0) 11) #f)
  (test (in-bounds? (make-posn 10 0) 11) #t)
  (test (in-bounds? (make-posn 0 10) 11) #f)
```

<*on-cats-path?-tests*> **::=**

```
(test ((on-cats-path? (make-world (empty-board 5)
                                  (make-posn 1 1)
                                  'playing
                                  5
                                  (make-posn 0 0)
                                  #t))
       (make-posn 1 0))
      #t)
(test ((on-cats-path? (make-world (empty-board 5)
                                  (make-posn 1 1)
                                  'playing
                                  5
                                  (make-posn 0 0)
                                  #f))
       (make-posn 1 0))
      #f)

(test ((on-cats-path? (make-world (empty-board 5) (make-posn 1 1)
                                  'playing 5 (make-posn 0 0) #t))
       (make-posn 2 1))
      #f)
(test ((on-cats-path?
        (make-world (list
                     (make-cell (make-posn 0 1) #t)
                     (make-cell (make-posn 1 0) #t)
                     (make-cell (make-posn 1 1) #f)
                     (make-cell (make-posn 1 2) #t)
                     (make-cell (make-posn 2 0) #t)
                     (make-cell (make-posn 2 1) #t)
                     (make-cell (make-posn 2 2) #t))
                    (make-posn 1 1)
                    'cat-lost
                    3
                    (make-posn 0 0)
                    #t))
       (make-posn 0 1))
      #f)
```

<*+/f-tests*> **::=**

```
(test (+/f '∞ '∞) '∞)
(test (+/f '∞ 1) '∞)
(test (+/f 1 '∞) '∞)
(test (+/f 1 2) 3)
```

<*render-world-tests*> **::=**

```
(test
 (render-world
  (make-world (list (make-cell (make-posn 0 1) #f))
              (make-posn 0 1)
              'playing
              3
              (make-posn 0 0)
              #f))
 (overlay
  (render-board (list (make-cell (make-posn 0 1) #f))
                3
                (lambda (x) #t)
                #f)
  (move-pinhole thinking-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))

(test
 (render-world
  (make-world (list (make-cell (make-posn 0 1) #f))
              (make-posn 0 1)
              'cat-won
              3
              #f
              #f))
 (overlay
  (render-board (list (make-cell (make-posn 0 1) #f))
                3
                (lambda (x) #t)
                #f)
  (move-pinhole happy-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))

(test
 (render-world
  (make-world (list (make-cell (make-posn 0 1) #f))
              (make-posn 0 1)
              'cat-lost
              3
              #f
              #f))
 (overlay
  (render-board (list (make-cell (make-posn 0 1) #f))
                3
                (lambda (x) #t)
                #f)
  (move-pinhole mad-cat
                (- (cell-center-x (make-posn 0 1)))
                (- (cell-center-y (make-posn 0 1))))))

(test
 (render-world
  (make-world (list
               (make-cell (make-posn 0 1) #t)
               (make-cell (make-posn 1 0) #t)
```

<*chop-whiskers-tests*> **::=**

```
(test (chop-whiskers (rectangle 6 6 'solid 'black))
      (put-pinhole (rectangle 3 3 'solid 'black) 0 0))

(test
 (pinhole-x
  (render-world
   (make-world
    (empty-board 3)
    (make-posn 0 0)
    'playing
    3
    (make-posn 0 0)
    #f)))
 0)
(test
 (pinhole-x
  (render-world
   (make-world
    (empty-board 3)
    (make-posn 0 1)
    'playing
    3
    (make-posn 0 0)
    #f)))
 0)
```

<*render-board-tests*> **::=**

```
(test (render-board (list (make-cell (make-posn 0 0) #f))
                    3
                    (lambda (x) #f)
                    #f)
      (overlay
       (nw:rectangle (world-width 3)
                     (world-height 3)
                     'solid
                     'white)
       (render-cell (make-cell (make-posn 0 0) #f)
                    #f
                    #f)))

(test (render-board (list (make-cell (make-posn 0 0) #f))
                    3
                    (lambda (x) #t)
                    #f)
      (overlay
       (nw:rectangle (world-width 3)
                     (world-height 3)
                     'solid
                     'white)
       (render-cell (make-cell (make-posn 0 0) #f)
                    #t
                    #f)))


(test (render-board (list (make-cell (make-posn 0 0) #f))
                    3
                    (lambda (x) #f)
                    #f)
      (overlay
       (nw:rectangle (world-width 3)
                     (world-height 3)
                     'solid
                     'white)
       (render-cell (make-cell (make-posn 0 0) #f)
                    #f
                    #f)))

(test (render-board (list (make-cell (make-posn 0 0) #f)
                          (make-cell (make-posn 0 1) #f))
                    3
                    (lambda (x) (equal? x (make-posn 0 1)))
                    #f)
      (overlay
       (nw:rectangle (world-width 3)
                     (world-height 3)
                     'solid
                     'white)
       (render-cell (make-cell (make-posn 0 0) #f)
                    #f
                    #f)
       (render-cell (make-cell (make-posn 0 1) #f)
                    #t
                    #f)))
```

<*render-cell-tests*> **::=**

```
(test (render-cell (make-cell (make-posn 0 0) #f) #f #f)
      (move-pinhole (circle circle-radius 'solid normal-color)
                    (- circle-radius)
                    (- circle-radius)))
(test (render-cell (make-cell (make-posn 0 0) #t) #f #f)
      (move-pinhole (circle circle-radius 'solid 'black)
                    (- circle-radius)
                    (- circle-radius)))
(test (render-cell (make-cell (make-posn 0 0) #f) #t #f)
      (move-pinhole (overlay (circle circle-radius 'solid normal-color)
                             (circle (quotient circle-radius 2) 'solid
                                     on-shortest-path-color))
                    (- circle-radius)
                    (- circle-radius)))
(test (render-cell (make-cell (make-posn 0 0) #f) #t #t)
      (move-pinhole (overlay (circle circle-radius 'solid normal-color)
                             (circle (quotient circle-radius 2) 'solid
                                     under-mouse-color))
                    (- circle-radius)
                    (- circle-radius)))
```

<*world-size-tests*> **::=**

```
(test (world-width 3) 150)
(test (world-height 3) 117)
```

<*cell-center-x-tests*> **::=**

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

<*cell-center-y-tests*> **::=**

```
(test (cell-center-y (make-posn 1 1))
      (+ circle-radius (* 2 circle-spacing 433/500)))
```

<*clack-tests*> **::=**

```
(test (clack
       (make-world '() (make-posn 0 0) 'playing 3 #f #f)
       1 1 "button-down")
      (make-world '() (make-posn 0 0) 'playing 3 #f #f))
(test (clack
       (make-world '() (make-posn 0 0) 'playing 3 #f #f)
       1 1 'drag)
      (make-world '() (make-posn 0 0) 'playing 3 #f #f))
(test (clack
       (make-world (list (make-cell (make-posn 0 0) #f))
                   (make-posn 0 1)
                   'playing
                   3
                   #f
                   #f)
       (cell-center-x (make-posn 0 0))
       (cell-center-y (make-posn 0 0))
       'move)
      (make-world
       (list (make-cell (make-posn 0 0) #f))
       (make-posn 0 1)
       'playing
       3
       (make-posn 0 0)
       #f))
(test (clack
       (make-world (list (make-cell (make-posn 0 0) #f))
                   (make-posn 0 1)
                   'playing
                   3
                   #f
                   #f)
       (cell-center-x (make-posn 0 0))
       (cell-center-y (make-posn 0 0))
       'enter)
      (make-world
       (list (make-cell (make-posn 0 0) #f))
       (make-posn 0 1)
       'playing
       3
       (make-posn 0 0)
       #f))
(test (clack
       (make-world '() (make-posn 0 0)
                   'playing 3 (make-posn 0 0) #f)
             1 1 'leave)
      (make-world '() (make-posn 0 0) 'playing 3 #f #f))

(test (clack (make-world '() (make-posn 0 0)
                         'playing 3 (make-posn 0 0) #f)
             10
             10
             "button-down")
      (make-world '() (make-posn 0 0) 'playing 3 #f #f))

(test (clack (make-world (list (make-cell (make-posn 0 0) #f)
```

<*update-world-posn-tests*> **::=**

```
(test (update-world-posn
       (make-world (list (make-cell (make-posn 0 0) #f))
                   (make-posn 0 1) 'playing 3 #f #f)
       (make-posn (cell-center-x (make-posn 0 0))
                  (cell-center-y (make-posn 0 0))))
      (make-world (list (make-cell (make-posn 0 0) #f))
                  (make-posn 0 1) 'playing 3 (make-posn 0 0) #f))

(test (update-world-posn
       (make-world (list (make-cell (make-posn 0 0) #f))
                   (make-posn 0 0) 'playing 3 #f #f)
       (make-posn (cell-center-x (make-posn 0 0))
                  (cell-center-y (make-posn 0 0))))
      (make-world (list (make-cell (make-posn 0 0) #f))
                  (make-posn 0 0) 'playing 3 #f #f))

(test (update-world-posn
       (make-world (list (make-cell (make-posn 0 0) #f))
                   (make-posn 0 1) 'playing 3 (make-posn 0 0) #f)
       (make-posn 0 0))
      (make-world (list (make-cell (make-posn 0 0) #f))
                  (make-posn 0 1) 'playing 3 #f #f))
```

<*move-cat-tests*> **::=**

```
(test
 (move-cat
  (make-world (list (make-cell (make-posn 1 0) #f)
                    (make-cell (make-posn 2 0) #f)
                    (make-cell (make-posn 3 0) #f)
                    (make-cell (make-posn 4 0) #f)

                    (make-cell (make-posn 0 1) #f)
                    (make-cell (make-posn 1 1) #t)
                    (make-cell (make-posn 2 1) #t)
                    (make-cell (make-posn 3 1) #f)
                    (make-cell (make-posn 4 1) #f)

                    (make-cell (make-posn 0 2) #f)
                    (make-cell (make-posn 1 2) #t)
                    (make-cell (make-posn 2 2) #f)
                    (make-cell (make-posn 3 2) #t)
                    (make-cell (make-posn 4 2) #f)

                    (make-cell (make-posn 0 3) #f)
                    (make-cell (make-posn 1 3) #t)
                    (make-cell (make-posn 2 3) #f)
                    (make-cell (make-posn 3 3) #f)
                    (make-cell (make-posn 4 3) #f)

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

                   (make-cell (make-posn 0 1) #f)
                   (make-cell (make-posn 1 1) #t)
                   (make-cell (make-posn 2 1) #t)
                   (make-cell (make-posn 3 1) #f)
                   (make-cell (make-posn 4 1) #f)

                   (make-cell (make-posn 0 2) #f)
                   (make-cell (make-posn 1 2) #t)
                   (make-cell (make-posn 2 2) #f)
                   (make-cell (make-posn 3 2) #t)
                   (make-cell (make-posn 4 2) #f)

                   (make-cell (make-posn 0 3) #f)
                   (make-cell (make-posn 1 3) #t)
                   (make-cell (make-posn 2 3) #f)
                   (make-cell (make-posn 3 3) #f)
                   (make-cell (make-posn 4 3) #f)
```

<*change-tests*> **::=**

```
(test (change (make-world '() (make-posn 1 1)
                          'playing 3 (make-posn 0 0) #f)
              "h")
      (make-world '() (make-posn 1 1)
                  'playing 3 (make-posn 0 0) #t))
(test (change (make-world '() (make-posn 1 1)
                          'playing 3 (make-posn 0 0) #t)
              "release")
      (make-world '() (make-posn 1 1) 'playing 3 (make-posn 0 0) #f))
```

<*point-in-this-circle?-tests*> **::=**

```
(test (point-in-this-circle? (make-posn 0 0)
                             (cell-center-x (make-posn 0 0))
                             (cell-center-y (make-posn 0 0)))
      #t)
(test (point-in-this-circle? (make-posn 0 0) 0 0)
      #f)
```

<*find-best-positions-tests*> **::=**

```
(test (find-best-positions (list (make-posn 0 0)) (list 1))
      (list (make-posn 0 0)))
(test (find-best-positions (list (make-posn 0 0)) (list '∞))
      #f)
(test (find-best-positions (list (make-posn 0 0)
                                 (make-posn 1 1))
                           (list 1 2))
      (list (make-posn 0 0)))
(test (find-best-positions (list (make-posn 0 0)
                                 (make-posn 1 1))
                           (list 1 1))
      (list (make-posn 0 0)
            (make-posn 1 1)))
(test (find-best-positions (list (make-posn 0 0)
                                 (make-posn 1 1))
                           (list '∞ 2))
      (list (make-posn 1 1)))
(test (find-best-positions (list (make-posn 0 0)
                                 (make-posn 1 1))
                           (list '∞ '∞))
      #f)
```

<*lt/f-tests*> **::=**

```
(test (<=/f 1 2) #t)
(test (<=/f 2 1) #f)
(test (<=/f '∞ 1) #f)
(test (<=/f 1 '∞) #t)
(test (<=/f '∞ '∞) #t)
```

<*circle-at-point-tests*> **::=**

```
(test (circle-at-point empty 0 0) #f)
(test (circle-at-point (list (make-cell (make-posn 0 0) #f))
                       (cell-center-x (make-posn 0 0))
                       (cell-center-y (make-posn 0 0)))
      (make-posn 0 0))
(test (circle-at-point (list (make-cell (make-posn 0 0) #f)
                             (make-cell (make-posn 0 1) #f))
                       (cell-center-x (make-posn 0 1))
                       (cell-center-y (make-posn 0 1)))
      (make-posn 0 1))
(test (circle-at-point (list (make-cell (make-posn 0 0) #f))
                       0 0)
      #f)
```

<*blocked-cells-tests*> **::=**

```
(test (block-cell (make-posn 1 1)
                  (list (make-cell (make-posn 0 0) #f)
                        (make-cell (make-posn 1 1) #f)
                        (make-cell (make-posn 2 2) #f)))
      (list (make-cell (make-posn 0 0) #f)
            (make-cell (make-posn 1 1) #t)
            (make-cell (make-posn 2 2) #f)))

(test (add-n-random-blocked-cells 0 (list (make-cell (make-posn 0 0)
                                                     #t))
                                  3)
      (list (make-cell (make-posn 0 0) #t)))
(test (add-n-random-blocked-cells 1 (list (make-cell (make-posn 0 0)
                                                     #f))
                                  3)
      (list (make-cell (make-posn 0 0) #t)))
```

1.18.10 Run, program, run

This section contains the main expression that starts the Chat Noir game going.

<*go*> ::=

```
(let* ([board-size 11]
       [initial-board
        (add-n-random-blocked-cells
         6
         (empty-board board-size)
         board-size)]
       [initial-world
        (make-world initial-board
                    (make-posn (quotient board-size 2)
                               (quotient board-size 2))
                    'playing
                    board-size
                    #f
                    #f)])

  (big-bang initial-world
            (on-draw render-world
                     (world-width board-size)
                     (world-height board-size))
            (on-key change)
            (on-mouse clack)
            (name "Chat Noir"))
  (void))
```

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.