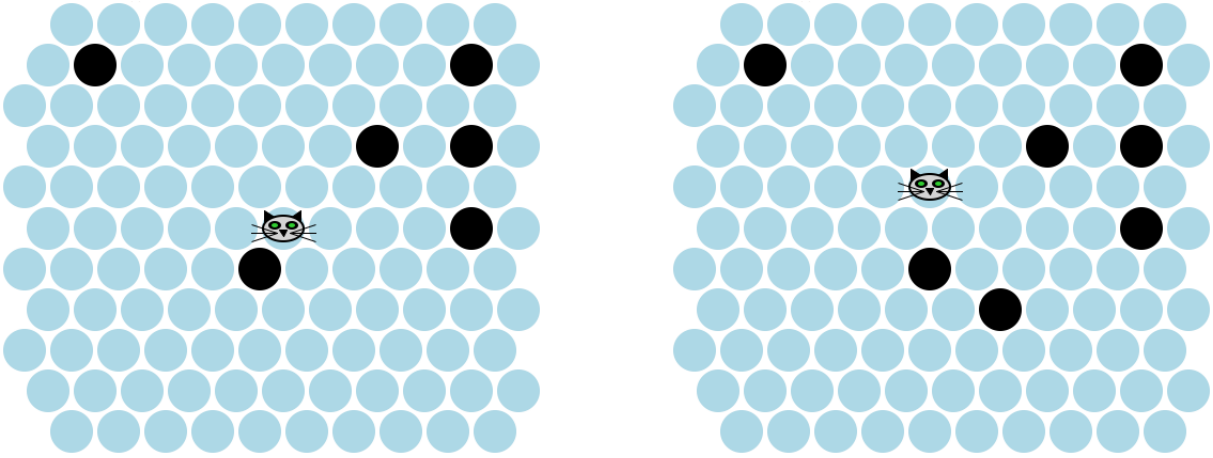
Project 3: CHAT NOIR

Due: December 8th



CHAT NOIR is a two player game played on a hexagonal grid. The grid has 119 spaces (as shown in the pictures above) and six randomly chosen spaces are blocked off. Initially a black cat is on the middle space. The "keeper" player goes first by clicking on a space (any space except one that is already blocked off or has the cat) to block it off. Then, the "chat" player responds by clicking on a space that is adjacent to the cat (and unblocked), causing the cat to move one step. The chat player wins when it reaches the edge of the board and the keeper player wins if the cat is unable to take a step (*i.e.*, all of the neighboring cells are blocked). The two images at the top of the page show an initial position and the board after two moves (first one for the keeper and then one for the cat).

The assignment is inspired by this online game, but is slightly different:

`http://www.gamedesign.jp/flash/chatnoir/chatnoir.html`

1 CHAT NOIR on a single computer

Your job is to implement CHAT NOIR, including designing the data structures to represent a state of the game, rendering the state into a board (something similar to the above), and accepting clicks on the squares to accommodate the players' moves.

Begin by thinking carefully about the data structures you use to represent the board. What information do you need to keep track of in order to play the game? Worry about cat and player movement first, drawing last.

Make a few examples of initial boards and then simulate a few turns by hand to make sure your data definitions work well (these examples will come in handy when designing test cases too, of course). The more time you spend at this stage understanding your data and working with it by hand, the fewer blind alleys you will run down when programming. Start by working with a 23 cell board (five by five) to make the examples more manageable.

Every other row in a hexagonal grid is lined up, and adjacent rows are off by 1/2 of the cell size in the horizontal direction. Also, the top and bottom rows in CHAT NOIR do not contain the left-most cell (since it is useless).

At this point you have designed about 90% of the complete program. To finish, you must create a two-player, *distributed* version of CHAT NOIR, that is, a game that two people can play against each other wherever they happen to be located. The rules of the game are as before, but now the cat will be under the control of a second player.

2 Distribution warmup

Designing a distributed program requires no more design knowledge than you already need to design world programs in ISL+, though it broadens the focus from the representation of the world to the representation of *two* worlds and the messages that pass back and forth between them. It also demands that you study the documentation of a new API, a common task for programmers.

Before we dive in and adapt CHAT NOIR to a distribute world, follow these steps to understand the basics of how distributed world programs are written.

1. Download and install the `universe2.ss` teachpack. (It replaces the `world.ss` teachpack, including all of the image functions, the world functions and some new functions.)

   `http://www.cs.uchicago.edu/~robby/courses/15100-2008-fall/universe2.ss`

2. Read the `universe2.ss` teachpack documentation (focusing on sections 6, 7, and 8), located at:

   `http://www.cs.uchicago.edu/~robby/courses/15100-2008-fall/universe/`

3. Study the design of the ball world.

As a warm-up exercise, tackle the design of a distributed program that passes mouse clicks from one computer to another and back. More precisely, each player runs a world program that opens a canvas of size 100 x 100 pixels. At any point in time, one world is active and the other one is passive. The active world contains a colored spot where the passive world experienced the last mouse click. The passive world is blank and ignores all mouse clicks. When the user of the active world clicks on the canvas, it sends a message with the mouse click's coordinates to the passive world causing a switch of roles.

When you are designing such a pair of collaborating programs, you must answer three questions:

1. *What happens at the beginning?* All we know is that the universe must start up first and that the two worlds follow in some unspecified order. Clearly, it is the universe's role to pick one of the two worlds and to tell it that it is active. By implication, the other world remains passive.

2. *What protocol do we use when the two worlds know their initial states?* Now that one world is active and the other one is passive, we just need to ensure that when a "player" clicks in the active world, the mouse click coordinates are sent to the passive world and that the world becomes passive.

3. *How does it all end?* For this simple universe, there is no end.

These answers have implications for the kind of messages that go back and forth as well as the state space of the two worlds. As far as the messages are concerned, we can clearly get away with just one kind of message if we are willing to accept that the universe picks a random mouse position at first:

```
;; A Msg is (list Nat Nat)
```

Question: Why can't we use a `Posn` instead? If you don't know the answer, please re-read the documentation of the `universe.ss` teachpack especially the section on messages and S-expressions.

From the documentation, we also know that the choice of messages narrows down the definition of `Mail` from the documentation:

```
;; A Mail is (listof (list Player Msg))
```

Look up `Player` in the documentation.

In turn, the data definition for `Mail` and the problem statement now determine the contracts for the functions we need to start a universe server:

```
;; create : Player Player -> (cons Universe Mail)
;; create the initial state and the first message
(define (create player1 player2) ...)

;; process: Universe Player Msg -> (cons Universe Mail)
;; p sent message m in universe u; pass m on to other player
(define (process u p m) ...)
```

Mini Exercise 1: Design the state space of the universe. If you need help, (re)read the section on designing a ball-throwing universe in the documentation. Once you have figured out the universe's state space, finish the design of `create` and `process`. Hint: Recall from the documentation that your program can't create `Players` and that they must compare the identify of players with `eq?`.

Warning: You may start the server with `(universe create process)` but don't attempt to do this until after you have finished the design of your functions. ∎

As far as the worlds are concerned, the relevant functions are `rec` and `clack`:

```
;; rec : World Msg -> World
;; turn Msg into the current posn of the world
(define (rec w msg) ...)

;; clack : World Nat Nat Mouse -> (make-package World Msg)
;; on button click, send mouse posn and ...
(define (clack w x y me) ...)
```

Again, their signatures are determined by our answers to the standard design questions and the (general versions in the) documentation.

Mini Exercise 2: Design the state space of the world(s). Keep in mind that it needs to reflect two distinct properties: which position to paint and whether it is active. Also recall that the world should be blank when it is passive. Then design the function `render`, which creates a scene from a given world. ∎

Mini Exercise 3: Finish the design of `rec` and `clack`. ∎

Finally, organize your code into four files using `require` and `provide`. The `provide` declaration accepts a sequence of names and declares that those names are exported from the current file. The `require` declaration accepts a single filename and imports all of the names that were exported by `provide` from that file.

The first file should contain nearly all of your program, including all of the functions defined above. Call this file `main.ss`. Add a `provide` line like this:

```
(provide rec clack create process)
```

Include the names of any other definitions that you need to start the universe. The remaining three files should each contain the line

```
(require "main.ss")
```

and should each be followed by calls to the various primitives in the `universe.ss` teachpack for starting the universe and the communicating worlds. Open each file in a separate tab and click run on each one to start up the universe (*i.e.* `universe`, `big-bang`, *etc.*). Start the universe first and then the worlds. (Note that you should include nearly all of the code in `main.ss`; the other three files will be much shorter.)

3 Distributed CHAT NOIR

You are now ready to complete the design of a distributed version of CHAT NOIR. Work through the same three questions that we used in the previous step. Design server-to-world messages and a universe server that initializes both worlds. (What must be the content of the first message from the server?)

Then design the worlds—specifically a state space and functions for receiving and sending mouse clicks—so that the two worlds can receive the initial messages from the CHAT NOIR universe and mouse-click messages from each other.

Finally, be sure that the image for the keeper world and the chat world look slightly different in some obvious way, so it is clear which is which when you're playing the game by yourself on your computer (`LOCALHOST`).

As much as possible, keep test cases together with the function they test.

The handin server only accepts one file per assignment. So please handin your `main.ss` file, and copy the contents of your other files to the bottom of that file (and comment them out). Clearly mark which parts belong in the other files, and the graders will separate them back out into four files. Use a format something like this (recall that `#|` and `|#` are comment delimiters):

```
#| server.ss
... code to start the server here
|#

#| chat.ss
... code to start start the chat player here
|#

#| keeper.ss
... code to start start the keeper player here
|#
```