1. Introduction. This document details the implementation of an API for the tron game used in CSCI-B351. The tron api is focused on providing a simple but flexible interface for allowing pluggable game brains or A.I.'s to play a tron game. In tron, two players compete against one another for space. Each player drives a virtual light cycle that continuously flies inside of a virtual map (2-dimensional) at a constant rate of speed. You can turn your cycle left and right through the map, trying to stay alive longer than your opponent.

The walls are deadly, and hitting any wall or protrusion results in instant death. Additionally, your light cycles are emitting light that leaves a solidified trail behind, and hitting these walls, either your own or the trail left by your opponent also results in instant death. Don't die first! You can either win the game by being alive longer than your opponent, lose by dying first, or you can draw by dying at the exact same time.

In this library, each cycle moves in discrete lock-step fashion, where the players each take a turn at the same time, moving their cycle forward one step. Since the most interesting aspects of the game are the actual brains that drive the cycles, this library provides all of the basic functionality for creating tron brains. Here's a simple list of the features that are provided:

1. You can define brains that handle the server protocol transparently, and use them both locally on your own machine and remotely to play on the server.
2. You can run your brain against others locally, so that you can test your brains offline without having to connect to the game server for each run.
3. You can easily run your brain against a remote server that you specify.

Every game starts in the same way, there is a map, and then there are starting positions for each of the players. Each player is given the starting position of the other player, as well as the size of the map and the list of which coordinates in the map are walls. This means that both players have perfect information of the playing field when they start. Namely, the player will have the following information available during each turn:

size	The size of the map.
walls	The walls and obstacles of on the starting map.
pos	The locations of the two cycles on the map.

In addition to this basic information, the `define-tron-brain` form lets the user maintain arbitrary state over time so that additional information that the user provides is available on each turn.

For those who wish to see how to use the main bindings of this library, `define-tron-brain` and `play-tron`, you can read through page 7. The rest of the document is available for those who wish to understand the entire source code. Users may also be interested in the **Board Utilities** section, which provides some useful procedures for working with the boards.

2. *The Board Representation.* The board on which the cycles are playing is a standard 2 dimensional grid that has $[0, w)$ columns and $[0, h)$ rows, where h is the height of the map and w is the width. The origin is in the upper lefthand corner. We refer to a given grid by coordinate point in the form `(x . y)` where `x` is the column and `y` is the row. The game is played on a torus board, meaning that if you go off the edge of the board on one side, you appear on the opposite side of the board going the same direction. Each cycle's state on the board is recorded as a coordinate in the above plane. A position is a pair `(x . y)` where `x` is the horizontal position of the cycle (or column) and `y` is the vertical (or row) position of the column. The indexes `x` and `y` are zero-indexed.

You can move on a board in any of the major compass direction. That is, you can move north, south, east, or west. You cannot move diagonally in a single step. The variable `valid-moves` is a list containing the valid directional moves in our symbolic representation.

```
(define valid-moves '(n w s e))
```

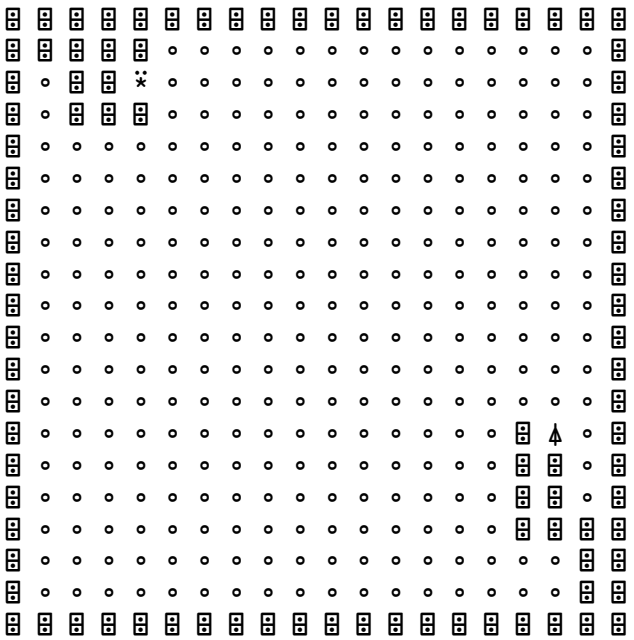
3. Tron Brains. To begin, let's get some basic terminology down. At the heart of this library is the concept of a tron *brain*. A tron brain can be thought of as the artificial intelligence that will drive a specific light cycle. A tron brain has one and only one job, to decide the next move to make, which is really what the next direction should be. We use a special syntax called `define-tron-brain` to actually create the brains. This syntax handles all of the heavy lifting in the background so that the code the user (that's you) has to write is limited to deciding the right move to make.

Since most people like to start with an example, let's consider one. As an example of using `define-tron-brain` let's make a brain that randomly plays a move. Our brain's name will be "Random move bot." This bot is smart enough to not run into any walls if it doesn't have to. On the other hand, if it doesn't have a safe move that it can make, it will choose a move to make at random from among all of the possible valid moves, even though it knows that none of them are safe, because you always must send a move to the server, even if it is a bad one. Pay attention to our use of `(walls orig-walls)` which sets up our state to be `orig-walls` initially. On each iteration, we update `walls` implicitly by returning the new value we want to be held by `walls`. We use this to keep track of the trails left behind by the cycles as well as the original walls. We will talk about this more in further sections when we talk about the syntax of `define-tron-brain` in detail.

⟨ Define `define-tron-brain` 6 ⟩

```
(define-tron-brain (random-move-bot
                    ("Random Move Bot" size orig-walls play ppos opos)
                    (walls orig-walls))
  (let ([new-walls (cons* ppos opos walls)])
    (define (safe? m)
      (and (not (member (get-pos m ppos size) new-walls)) #t))
    (let ([safe-moves (filter safe? valid-moves)])
      (play
        (if (null? safe-moves)
            (list-ref valid-moves (random (length valid-moves)))
            (list-ref safe-moves (random (length safe-moves)))))
      new-walls)))
```

4. Here's an example of the random brain in action, using the printed representation from `play-tron`, which is described later. Here, the first player is represented by ▲ and has won the game, whereas the ⁘ is the crashing, smoldering remains of what was player 2 (▲). The dominos ⚅ are what is left over from the trail of their cycles.



5. Creating brains. The syntax for defining a tron brain is as follows:

```
(define-tron-brain (proc (name size walls play ppos opos)
                         (state init))
  body+ ...)
```

Now, there are quite a few identifiers there, so it's helpful to start with what the user would normally write. Basically, the `body+ ...` code is the code that will actually be run each time that a brain is needed to get a specific move for a cycle. That is, if we are simulating a game locally using `play-tron` (discussed later), then every time we need to know what the next move is going to be, the `play-tron` procedure will evaluate the `body+ ...` code. That `body+ ...` code had better send the next move to the server setup by `play-tron`.

The `define-tron-brain` syntax provides a lot of assistant to the user so that most of the information that one would need to decide what move to make next is readily available. Let's start with a short synopsis of what each of the identifiers in the `define-tron-brain` syntax will be when we run the `body+ ...` code.

Identifier	Use
proc	Name of brain procedure
name	String of player name
state	Variable to hold user provided state
init	Initial state expression
size	Holds the size of the map
walls	List of coordinates for walls
ppos	Player position
opos	Opponent position
play	Procedure to play move
body+	One or more expressions for the brain

The `name` element should be an expression that evaluates into a string that will be used as the name of the player on the server when reporting scores and information. The `size` variable is a pair `(w . h)` containing the width and height of the map. The `walls` variable will contain a list of the form `((x . y) ...)` that gives the locations of each wall on the map. The variables `ppos` and `opos` both have the form `(x . y)` and represent the current positions of the player and opponent respectively.

The `proc` identifier that you give will be the only definition that is actually visible outside of the `body+ ...` code. It will be bound to a procedure that represents the brain you have just defined. When we talk about the main brain procedures or objects we are talking about the procedures `proc` that are defined by `define-tron-brain`. You need to make sure that any simulator that you use such as `play-tron` that needs a brain is given one or more of these procedures. You can read more about `play-tron` further down to know exactly what it expects to be given when you run it.

We haven't yet talked about `play`. This is an interesting one. Specifically, `play` is a procedure that is visible in `body+ ...` that you use to send a move to the server. That is, if you want to send a move such as north to the server, you don't need to do anything fancy: all you do is evaluate something like `(play 'n)` before the end of `body+ ...` and you will be done. You should be careful to only call `play` once for each invocation of your brain. You shouldn't send more than one move at a time to the server. The `play` procedure has the following signature:

$$\texttt{play} : \textit{move} \rightarrow \texttt{\#<void>}$$

Now let's talk about the state. It may be that the brain needs to keep some information around each time that it is called, such as a history of the moves that have already been made. To facilitate this, we allow the user to specify a variable `state` that will be initialized to the value of the `init` expression, which is evaluated once for that value. Then, on each invocation of the brain mover, when we evaluate the `body+ ...` code, we will take the value returned by the `body+ ...` code and bind it to the `state` variable,

replacing the previous state. This means that the next time the brain mover (an internal element of the implementation of a brain that you don't have to worry about) is called, it will know what it returned last time. As you saw in the example, this can be used to keep track of the changing board state, and other interesting things.

At this point, if you do not care to see how we implement this macro, you can skip to the next titled section.

6. The `define-tron-brain` form binds `proc` to a procedure that can be used to initialize a simulation. The `proc` name will be bound to a procedure that expects to receive two arguments, which are ports, that allow it to communicate its moves to the driver, either locally or remotely.

$$\texttt{proc} : \textit{play-port} \times \textit{info-port} \rightarrow (\textit{port} \rightarrow \texttt{\#<void>})$$

The value returned by the `proc` procedure is another procedure which is the main brain procedure. Whenever we call this procedure, we expect that the user provided code will run, and that a single play will be made and sent to the server. This procedure handles the state itself, and so the state is not a part of the visible interface, hence the `#<void>` return and the single port argument. The port argument should allow us to get the player positions from the server (see the server protocol section for more details on this). We use a port here instead of passing `ppos` and `opos` directly because we need to be able to use the same brain on the remote server as well as a local one. The `proc` procedure does the initial protocol negotiation and gets all of the static information set up before returning the brain move making procedure.

The macro that implements all of this is a simple `syntax-rules` macro. There are a few things we do have to make sure about. Firstly, we need to be careful not to use `name` in more than one place, since it could be an expression and not a regular identifier. Secondly, we want to make sure that we make as many things visible in the bindings of our `init` expression as we can, since this makes it possible to use some values like `walls` to populate the initial state. We do this in the above example brain that moves randomly. We also want to remember to wrap the body `b1 b2 ...` in a let-nil and not a `begin` so that the user can provide definitions at the top of the form if they want to.

⟨ Define `define-tron-brain` 6 ⟩ ≡

```
(define-syntax define-tron-brain
  (syntax-rules ()
    [(k (proc (name size walls play ppos opos)
              (state init))
        b1 b2 ...)
     (define (proc play-port info-port)
       (format play-port "~a~n" name)
       (let ([play (make-play-proc 'play play-port)])
         (let* ([size (read info-port)] [walls (read info-port)])
           (let ([state init])
             (lambda (port)
               (let* ([ppos (read port)] [opos (read port)])
                 (let-values ([(newstate) (let () b1 b2 ...)])
                   (set! state newstate)))))))))]))
```

This section exports define-tron-brain.

This code is used in section 3.

7. Playing a game. To play a game locally, without having a connection to a server or anything like that, you use the `play-tron` procedure.

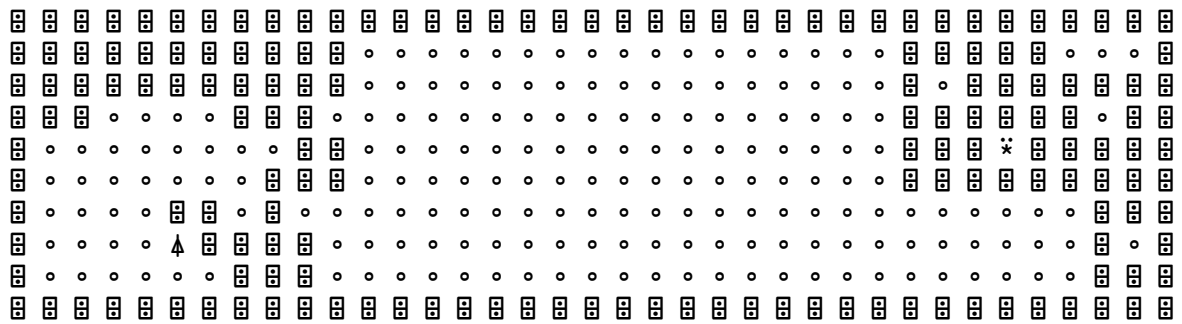
`play-tron` : *size* × *walls* × *player1-position* × *player2-positions* × *brain1* × *brain2* → `#<void>`

It expects to receive a board specification and two brains as defined by `define-tron-brain`. It will simulate the game and show the progress of the game on the terminal.

A board specification is composed of the size of the board, the walls or obstacles that are there at the start of the game, and the positions of the two cycles on the board. The two cycles should not be on any walls already (if you want the game to go for more than zero turns) and you probably want to make sure that the board you have is a fair game.

The two brains that you pass should both be procedures defined using the `define-tron-brain` syntax. Here's an example of using `play-tron` together with the last bit of output.

```
> (play-tron '(37 . 10) (make-outer-walls '(37 . 10)) '(1 . 1) '(35 . 8)
random-move-bot random-move-bot)
...
```



```
Random Move Bot (Player 1) has won
```

At this point you can safely pick whatever section you want to go to that interests you. The rest of this document provides more details about the implementation and the protocols, as well as some useful utilities that you may be interested in.

The `play-tron` procedure will take care of setting up a virtual server loop for the brains. It uses the same server protocol as that of the remote server, so the same brains will work on the local simulator as they will on the remote server. Playing a game locally follows the following basic operations underneath:

1. Setup the pseudo-server.
2. Run brains to do the initial protocol negotation and get the brain movers back.
3. Play the turns and print the results of each turn.
4. Print the result of the game when the game has been won or a draw has been reached.

Internally we use string ports instead of network ports to simulate the protocol of the game. We don't necessarily have a single, constant set of ports to work with, as we need to create new ports each time that we make a turn, which will be discussed in more detail later.

```
(define (play-tron size walls pos1 pos2 b1 b2)
  ⟨ Verify play-tron arguments 12 ⟩
  (let-values ([(b1-play-port b1-get) (open-string-output-port)]
               [(b2-play-port b2-get) (open-string-output-port)])
    (let ([ip1 (make-info-port size walls)]
          [ip2 (make-info-port size walls)])
      (let ([b1-play (b1 b1-play-port ip1)]
            [b2-play (b2 b2-play-port ip2)])
```

```
    (let ([name1 (parse-name (b1-get))] [name2 (parse-name (b2-get))])
      (printf "Name1: ~a~nName2: ~a~n" name1 name2)
      ⟨ Simulate tron game 8 ⟩)))))
```

8. Simulating the actual game loop is step 4 in the server protocol. We want to print the current state of the game, and then exit if we have won. In this case, we will fudge a bit on the protocol and not send the results to the two players, since we don't need the results handled by the brains individually. We have a simple time delay to make sure that we can actually watch the game in action as it is playing. We will create new ports each time to represent the player positions, rather than trying to maintain one single port for each player. We should make sure that we keep track of the tails of each of the cycles by adding them to the `walls` on each iteration.

⟨ Simulate tron game 8 ⟩ ≡

```
(let loop ([pos1 pos1] [pos2 pos2] [walls walls])
  (print-board size walls pos1 pos2)
  (let ([status (game-status pos1 pos2 walls)])
    (case status
      [(draw p1 p2) ⟨ Print game result 10 ⟩]
      [else
        (sleep (make-time 'time-duration 250000000 0))
        (let ([port1 (make-pos-port pos1 pos2)]
              [port2 (make-pos-port pos2 pos1)])
          (b1-play port1) (b2-play port2)
          (loop (get-pos (string->move (b1-get)) pos1 size)
                (get-pos (string->move (b2-get)) pos2 size)
                (cons* pos1 pos2 walls)))])))
```

This section captures pos1, pos2, size, walls, b1-play, b2-play, b1-get, b2-get, name1, and name2.

This code is used in section 7.

9. The `game-status` procedure takes the position of the first player, the position of the second player, and the current list of walls. It returns a symbol indicating the status of the game, one of `draw`, `p1`, `p2`, or `not-over`. The symbols `p*` indicate that the given player has one, and `draw` indicates that the game was a draw. If the game is not yet finished, meaning that both cycles are still alive, then the `game-status` procedure will return `not-over`.

$$\texttt{game-status} : \mathit{pos1} \times \mathit{pos2} \times \mathit{walls} \to \mathit{status}$$

```
(define (game-status pos1 pos2 walls)
  (let ([p1-dead? (member pos1 walls)]
        [p2-dead? (member pos2 walls)])
    (cond
      [(and p1-dead? p2-dead?) 'draw]
      [p1-dead? 'p2]
      [p2-dead? 'p1]
      [else 'not-over])))
```

10. When the game has concluded, we want to print the results of the win. The user should receive a nice message with both the player slot, either first or second position, and the name of the player brain.

⟨ Print game result 10 ⟩ ≡
```
(case status
  [(draw) (printf "The game was a draw.~n")]
  [(p1) (printf "~a (Player 1) has won!" name1)]
  [(p2) (printf "~a (Player 2) has won!" name2)]
  [else (error #f "invalid status" status)])
```
This section captures status, name1, and name2.

This code is used in section 8.

11. At this point it makes sense to specify what exactly we mean by things like the size of the board, or the walls. A valid size should be a pair `(w . h)` where $w > 0$ and $h > 0$. Walls should be a list of `((x . y) ...` where for each position, $x \in [0, w)$ and $y \in [0, h)$. This is the same restriction for any valid position in the board, such as the player positions.
```
(define (valid-size? size)
  (and (pair? size)
       (integer? (car size)) (integer? (cdr size))
       (positive? (car size)) (positive? (cdr size))))

(define (valid-walls? size walls)
  (and (list? walls)
       (for-all (lambda (pos) (valid-position? size pos)) walls)))

(define (valid-position? size pos)
  (and (pair? pos)
       (integer? (car pos)) (integer? (cdr pos))
       (<= 0 (car pos)) (<= 0 (cdr pos))
       (< (car pos) (car size)) (< (cdr pos) (cdr size))))
```

12. We use the above to provide decent error messages to the user when the `play-tron` procedure is given the wrong things.

⟨ Verify `play-tron` arguments 12 ⟩ ≡
```
(unless (valid-size? size)
  (error 'play-tron "invalid board size" size))
(unless (valid-walls? size walls)
  (error 'play-tron "invalid walls list" walls))
(unless (valid-position? size pos1)
  (error 'play-tron "invalid cycle position" pos1))
(unless (valid-position? size pos2)
  (error 'play-tron "invalid cycle position" pos2))
(unless (procedure? b1)
  (error 'play-tron "invalid brain" b1))
(unless (procedure? b2)
  (error 'play-tron "invalid brain" b2))
```
This section captures size, walls, pos1, pos2, b1, and b2.

This code is used in section 7.

13. The Server Protocol. The general process of a client making a connection to the tron server can be outlined thus:

1. Client sends player name.
2. Server sends the size of the map as `(w . h)`.
3. Server sends the walls as an a list `((x . y) ...)`.
4. Server sends the positions of the cycles, listing first the player and then the opponent.
5. Player sends move in the form of a direction.
6. If the game is won or a draw, the server sends `win` to the winner, and `loss` to the loser or `draw` to both in the case of a draw.
7. Otherwise, repeat starting at step 4.

In the first step, the player name should be sent as a single line with an end of line terminator. There is no special syntax here, so any characters are reasonably allowed, though not all clients may be able to support this. The size of the map is actually a Scheme datum, and will be treated as such. This holds true for the walls list as well as the positions of the cycles. The moves that are sent should all be symbols from the set `valid-moves`.

It should be noted that there is very little information to deal with error situations right now.

14. Let's define a procedure `parse-name` that will handle the initial data that reads the name from the client. All we have to do is read the name from the first line of the client and strip the trailing and leading whitespace.

```
(define (parse-name str)
  (define first-line
    (with-input-from-string str
      (lambda () (get-line (current-input-port)))))
  (define (strip lst)
    (or (memp (lambda (x) (not (char-whitespace? x))) lst)
        '()))
  (list->string
    (reverse (strip (reverse (strip (string->list first-line)))))))
```

15. We use the `make-play-proc` to create our procedures that will send the messages to the server. They create a `play` procedure that is closed over a particular port. In this case, since this is a forward or user facing procedure that we are returnning, we want to do some reasonable error checking here and report invalid moves that are sent to the server on the client side.

$$\texttt{make-play-proc} : \mathit{name} \times \mathit{port} \to (\mathit{move} \to \texttt{\#<void>})$$

We use the *name* value when reporting errors, but it is otherwise unnecessary for the actual computation. We will also ensure that we actually flush the buffer after we send our data, because we cannot be sure that the port we are given will actually flush the data in the way that we expect. At any rate, we make sure to flush and to put a newline at the end of our line for completeness.

```
(define (make-play-proc name port)
  (lambda (move)
    (unless (memq move valid-moves)
      (error name "invalid move" move))
    (format port "~s~n" move)
    (flush-output-port port)))
```

16. In the simulation of a tron game, because we are simulating the actual behavior of the server, we use using a number of different ports, where the real server might use only one for all of the communication. The first of these is our information port. This port should contain the board size and the wall information in the same format as the server will send it. In this case, this means just using `write` a few times.

```
(define (make-info-port size walls)
  (open-string-input-port
    (format "~s~s~n" size walls)))
```

17. Position ports encode the first part of the fourth protocol stage, where we send the player positions to each player. In this case, we assume that the first argument is the first player to send, and the second argument the opponent and second position to send.

```
(define (make-pos-port p1 p2)
  (open-string-input-port
    (format "~s~s~n" p1 p2)))
```

18. The procedure `string->move` should take a single string representing the play made by a player in the syntax of the server protocol. That is, it should be a symbol that is a valid member of `valid-moves`.

```
(define (string->move str)
  (let ([val (with-input-from-string str read)])
    (or (and (symbol? val) (member val valid-moves) val)
        (error #f "Not a valid move" val))))
```

19. Running on a server. In this version, we do not yet have support for making a connection to the remote server, but the general API design has been completed. When connecting to a remote server, all you need is a remote server, which is basically a server name and a port number, together with the brain that you want to use.

`play-remote-tron` : *host* × *port* × *brain* → `#<void>`

The `play-remote-tron` procedure should not return any specific value, but it should print something out when the game has either been lost, won, or drawn. In the current design for the protocol, you have no control over who you play.

```
(define (play-remote-tron host port brain)
  (error 'play-remote-tron "remote play not implemented"))
```

20. Board Utilities. In this section we discuss some of the useful board related procedures that you may want to use in your own programs. They handle things related to the display of the board and manipulating or getting information about boards.

21. First up, it's nice to be able to get a less sparse representation of the board for printing. The `print-board` procedure does the work of printing a nice representation of the board to the current output port.

`print-board` : *size* × *walls* × *position* × *position* → `#<void>`

```
(define (print-board size walls pos1 pos2)
  (let ol ([i 0])
    (unless (= i (cdr size))
      (let il ([j 0])
        (unless (= j (car size))
          ⟨ Print representation for position 22 ⟩
          (il (1+ j))))
      (newline)
      (ol (1+ i))))
  (newline))
```

22. We use Unicode APL symbols to represent each particular element of the board. This means that you should have a proper, unicode enabled terminal that can print these characters. We like these characters because they look neat and different, and they are rich enough to enable easy visual recognition during the game. Here's the mapping from Symbol to meaning.

Symbol	Usage
∘	Regular, unoccupied square
⌸	Wall or cycle trail
⍋	Player 1 cycle
⍒	Player 2 cycle
⁘	Dead cycle

⟨ Print representation for position 22 ⟩ ≡

```
(define p (cons j i))
(cond
  [(member p walls)
   (if (or (equal? p pos1) (equal? p pos2))
       (printf "⁘ ")
       (printf "⌸ "))]
  [(equal? p pos1) (printf "⍋ ")]
  [(equal? p pos2) (printf "⍒ ")]
  [else (printf "∘ ")])
```

This section captures walls, i, j, pos1, and pos2.

This code is used in section 21.

23. We provide the **get-pos** procedure, which takes a move or direction and an old position, and returns a new position assuming that we moved in that new direction.

`get-pos` : *move* × *position* × *size* → *new-position*

The **size** argument should be the size of the board. This procedure is useful if you want to explore possibilities, and you can see how we use it in the **random-move-bot**.

```
(define (get-pos m old size)
  (let ([width (car size)] [height (cdr size)])
    (let ([x (car old)] [y (cdr old)])
      (case m
        [(n) (cons x (mod (-1+ y) height))]
        [(w) (cons (mod (-1+ x) width) y)]
        [(e) (cons (mod (1+ x) width) y)]
        [(s) (cons x (mod (1+ y) height))]
        [else (error #f "invalid move" m)]))))
```

24. All tron boards without walls are just empty torus planes. It is common to want to put walls around the edges of the board as a starting point. If you want to do that, then you can get a list of the positions that you will need by using the **make-outer-walls** procedure here.

`make-outer-walls` : *size* → *outer-walls-list*

We don't use the most effecient implementation, but it's not the worst either, and it's easy to understand.

```
(define (make-outer-walls size)
  (let ([x (car size)]
        [y (cdr size)])
    (append
      (map (lambda (v) `(,v . 0)) (iota x))
      (map (lambda (v) `(,v . ,(sub1 y))) (iota x))
      (map (lambda (v) `(0 . ,v)) (iota y))
      (map (lambda (v) `(,(sub1 x). ,v)) (iota y)))))
```

25. The **fair-game?** procedure is a simple utility that helps you determine whether a given board you are creating is actually a fair game for either player, or whether it actually give the edge to one player or another. To do this, we are basically checking for symmetry of the board along the axis perpendicular to the axis running straight through both cycles.

`fair-game?` : *size* × *walls* × *player1-position* × *player2-position* → `boolean`

```
(define (fair-game? size walls pos1 pos2)
  (error 'fair-game? "not implemented yet"))
```

26. Quick Reference. The following is a set of quick references for those who are already familiar with the Tron API, but need a quick refresher of the major programming elements.

Defining brains. Used for creating brain procedures that are passed to `play-tron`.

```
(define-tron-brain (proc (name size walls play ppos opos)
                         (state init))
  body+ ...)
```

Playing a game. Used to simulate two brains playing against each other on the server, but locally instead. Does not require server connection.

```
(play-tron board-size board-walls
  player1-position player2-position
  player1-brain    player2-brain)
```

27. Index.

⟨ Define `define-tron-brain` 6 ⟩ Used in section 3.
⟨ Print game result 10 ⟩ Used in section 8.
⟨ Print representation for position 22 ⟩ Used in section 21.
⟨ Simulate tron game 8 ⟩ Used in section 7.
⟨ Verify `play-tron` arguments 12 ⟩ Used in section 7.

Tron API

(Version 1.0)