Gamesman Worksheet: Start thinking about your final project

The goal of this worksheet is to have you design the framework for the Gamesman final project. There is NO coding involved in this part of the assignment. ☺

Exercise 1: Representing the position

A *position* in Gamesman is a unique way of representing the state of the game being played. For any game, the position must contain enough information to determine two things: where all the pieces on the game board are and whose turn it is.

To better explain this, let's look at an example from example from *1,2,…10*. In this game a position is represented by a two-element sentence. The first word determines whose turn it is. The letter `l` represents `Left` player's turn, while the letter `r` represents `Right` player's turn. The second element represents the game board. In this case all the necessary board information is contained by a non-negative integer describing how many pieces are currently down on the game board.

Thus, a sample position might look like this: `(l 7)` and be drawn in Gamesman as shown below.

Thus, it is now `Left` player's turn and there are currently seven pieces on the game board.

```
|||||||||||||||||||||
0  1  2  3  4  5  6  7  8  9  10
```

In *Tomorrow's-Tic-Tac-Toe*, a position is represented as a sentence of elements. The first element of the sentence, represented by an `x` or an `o`, determines whose turn it is, just like in *1,2,…,10*. The `x` stands for `X-player` and the `o` stands for `O-player`. The following elements represent the rows of the *Tomorrow's-Tic-Tac-Toe* game board. Each row is made up of the symbols representing either game pieces or empty slots. Here, an `x` or an `o` represents an actual game piece and `-` represents an empty slot.

A sample position in *Tomorrow's-Tic-Tac-Toe* might appear as follows: `(x o-o x-- x--)`

The first element of this position indicates that it is now `X-player`'s turn. The rest of the position consists of words, each of which is a row. Every row contains characters, each of which represent either a game piece or an empty slot. This position indicates that both `X-player` and `O-player` have two pieces on the board. The `X-player` has one piece at position a1 and another at position a2; `O-player`, at positions a3 and c3. See Figure 1, Figure 2, and Figure 3 on the top of the following page for further examples of various positions and their text based graphical representation.

```
3  |  o - o
2  |  x - -
1  |  x - -
      a b c
```

Figure 1
(o o-o- xxox x-xo)

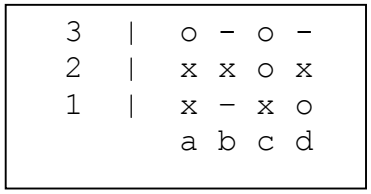


Figure 2
(x oxo o-- xox ---)

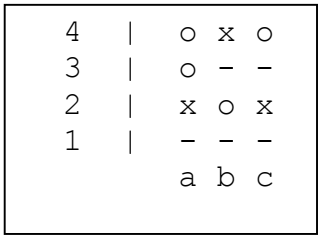
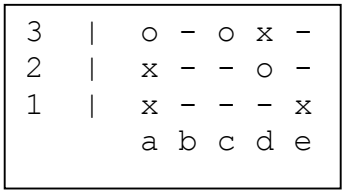


Figure 3
(o o-ox- x--o- x---x)



Your other handout describes all the games you can choose from in detail. On this handout you should see the recommended position representation to use for each game. This representation must have the ability to uniquely *map* to every possible instance in any game, including information about whose turn it is and where all the game pieces are. When you choose a game you are allowed to deviate from this suggested representation if you wish. You have the freedom to choose any *mapping* you wish, because Gamesman doesn't care how you represent positions. The important thing is that *you*, the programmer, understand it. Keep in mind that you must be able to tell, just by looking at the position, whose turn it is and where all of the pieces lie on the board. You will provide all the tools that gamesman needs to play and even solve your game.

a) Based on your experience playing all four new games in lab, choose the game you wish to code for this project. **Write the name of the game in the space below.** Now consider what your mapping of this board will be. Your *mapping* may be similar to that of *1,2,...,10* or *Tomorrow's-Tic-Tac-Toe*, or it can be entirely unique. It should be as intuitive as possible. Look at **`boards.pdf`** (or the handout from lecture) for the description of your game.

I will work on this game: _____.

My representation for *a fun position mid-way through the game is*: _____.
(Draw what that board would look like in the rectangle below – what you should focus on is making sure you understand the *mapping* between your chosen position, whose turn it is, and what the board looks like.)

Draw your board for (a) here…

…and it's _____'s turn.

b) Now that you've come up with a *mapping,* let's test its versatility. You will be required to write a function called `whose-move?`, which takes one argument, a position, and returns a word representing the name of the player whose turn it is to move. On the next page are two sample calls from *1,2,…,10*:

```
> (whose-move? '(r 5))    => Right

> (whose-move? '(l 2))    => Left
```

And now, two sample calls from *Tomorrow's Tic-Tac-Toe*:

```
> (whose-move? '(x x-- x-- o-o))    => X-player

> (whose-move? '(o x-- ox- x-o))    => O-player
```

Assume that you have already written `whose-move?` for your game. Fill in the first blank below with your answer from Part A and provide what your `whose-move?` function will return. The return value may include capital letters if you wish.

```
> (whose-move? _____________________________)

==>_____________________________________
```

Assume that the EXACT same position as above was given to you, except that it was the opposite player's turn. What would this position look like in your representation? Fill in this *new* position below and the return value from `whose-move?`.

```
> (whose-move? _____________________________)

==>_____________________________________
```

c) Now that you've come up way to access whose turn it is, let's figure out how to access the information about the game board. You will be required to write a function called `get-board` taking one argument, a position, and returning your game board representation. Below are two sample calls from *1,2,…,10*:

```
> (get-board '(r 5)) => 5

> (get-board '(l 2))    => 2
```

And now, two sample calls from *Tomorrow's Tic-Tac-Toe*:

```
> (get-board '(x x-- x-- o-o))   => (x-- x-- o-o)

> (get-board '(o x-- ox- x-o))   => (x-- ox- x-o)
```

d) Assume that you have already written `get-board` for your game. Fill in the first blank below with your answer from Part A and provide what your `get-board` function will return.

```
> (get-board _____________________________________________)

==>_____________________________
```

e) Assume that the EXACT same position as above was given to you, except that it was the opposite player's turnWhat would this position look like in your representation? Fill in this position below and the return value from `get-board`.

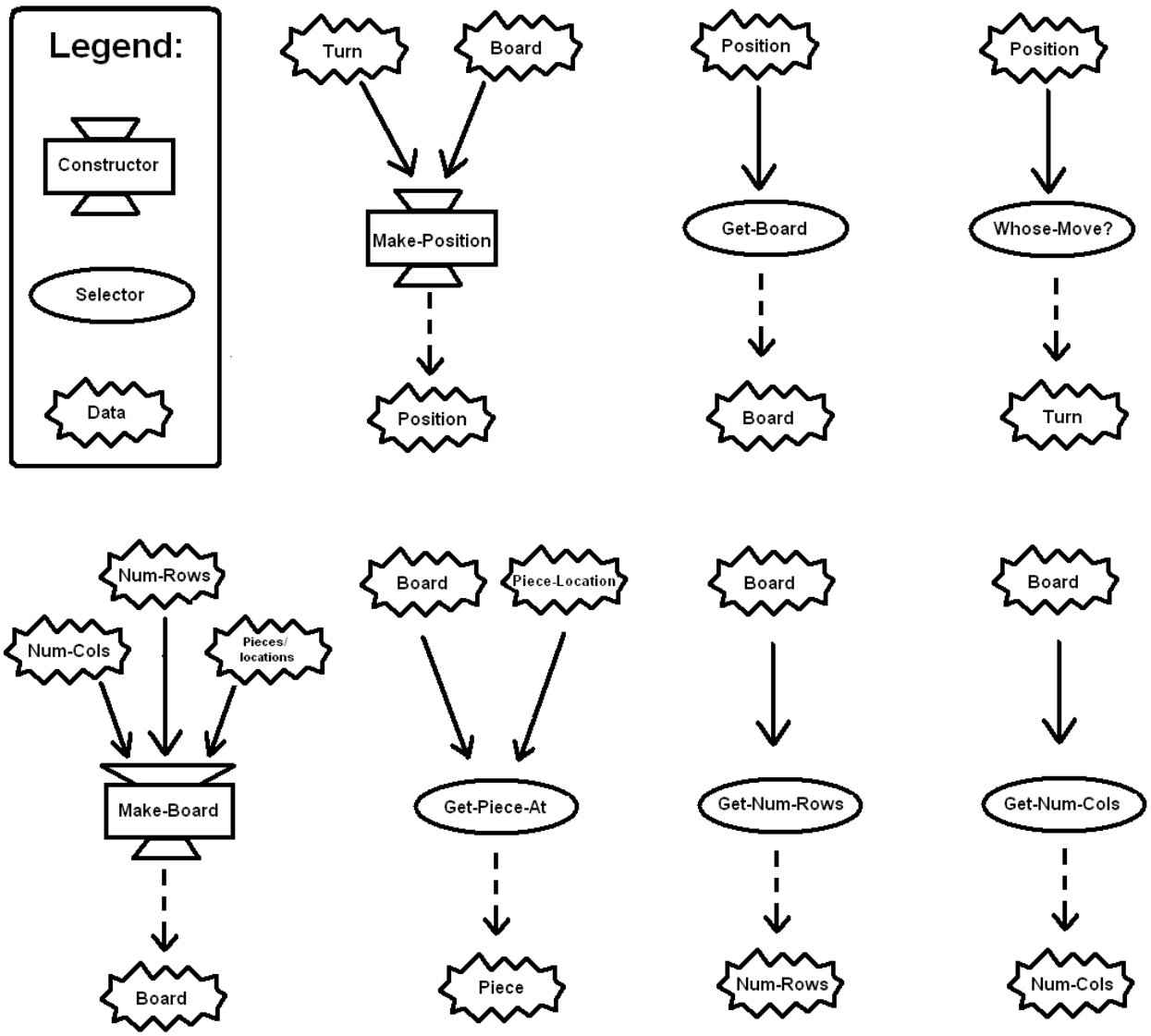
```
> (get-board _____________________________)

==>_____________________________
```

Exercise 2: Thinking Abstract…

Now you have an understanding of `get-board` and `whose-move?`, it is time to think about those functions in more abstract terms. `Get-board` and `whose-move?` each *select* specific information from the position. Thus, in abstract terms, these functions are called selectors.

The opposite of a selector, in abstract terms, is a constructor. You must *construct* the position from a turn and board. Likewise, the board and turn must also be *constructed* from their components. The turn is simple enough with little variance (one of 2 letters), so we do not require a constructor function for this, although you may choose to create one anyway. On the other hand, a board may vary in size with every game and it certainly changes state with every move of a player.

It is important to have a specific constructor for a board, and even selectors for the board. The following exercises will guide you in this endeavor. Refer to the drawing on the next page for a visualization of the constructor/selector concept with respect to positions and boards. Keep in mind that these diagrams are **general** and you may have to create **additional** constructors and selectors for your specific implementation. You will be required to implement the abstract functions in the diagrams below. These tools, along with the **additional** tools you decide to create will make the more complicated functions much easier to conceptualize as well as keep your program readable, both for you and your grader.



Exercise 3: Creating a board and extracting information from your board.

A *board* in gamesman is a representation of the location of game pieces on a game board at a particular instance in the game. The *board*, unlike a position, does not contain the information concerning whose turn it is. The *board* can have different representations for different games. Think about 1,2,…10 vs. Tomorrow's Tic-Tac-Toe for a moment. In the latter, the number of pieces on the game board increases with each move. Most slots could be empty or most could be full, depending on how long the game has been played. Information about each slot is carried by the *board*, whether it is empty or full. This requires you keep track of every slot on your game board, and since board size is arbitrary, this could change with every game. In some games, however, there are always a fixed number of game pieces on the game board and only their *location* changes with each move. For these games, it may be simpler to represent the *board* by keeping track of only the game board dimensions and the location of the game pieces. The advantage of this is the size of the position is independent of the game board size. The point is that there are many ways to represent a game boards for each game and store them within your position. Choose a representation that makes the most sense to you and is easiest to work with.

a) Based on the diagram on the previous page, `make-board` takes necessary input and *constructs* the game board. These inputs may differ slightly for each students' representation, even if they are coding the same game. The following examples call `make-board` for *Tomorrow's Tic-Tac-Toe.* Beneath each call to `make-board` is a picture of what the newly constructed board looks like. As you can see, it is necessary to have a good understanding of your chosen mapping.

```
> (make-board 3 4 '(a1x a2o a3o b2x c1x c2o c3o d1o d2x))
```

resulting board:

```
3  |  o - o -
2  |  o x o x
1  |  x - x o
      a b c d
```

```
> (make-board 3 3 '(a1x a2x a3o b2x c3o))
```

resulting board:

```
3  |  o - o
2  |  x x -
1  |  x - -
      a b c
```

b) Fill in the blanks below so that the call to `make-board` results in **your** board from Exercise 1, part D.

(num-rows) (num-cols) (pieces/locations)

```
> (make-board __________ _______ ____________________)

==> (board from Exercise 1D)
```

c) Once you can create your *board* you must also be able to select the relevant information about it, such as the number of rows, columns, or the contents of a particular place on the board. Below are some sample calls from *Tomorrow's Tic-Tac-Toe*:

```
> (get-num-rows (get-board '(x -x-o oo-x -x--))   ==> 3

> (get-num-cols (get-board '(o ox-o oo-x -x--))     ==> 4

> (get-piece-at (get-board '(o ox-o oo-x -x--)) 'b2) ==> o
```

c) Assume that you have already written the selectors `get-board`, `get-num-rows`, `get-piece-at`, and `get-num-cols` for your game.

Fill in the first blank below with your board from Exercise 1, part D and provide what your `get-num-rows` selector will return.

```
> (get-num-rows (get-board ____________________________________))

==>_____
```

Fill in the first blank below with your position from Exercise 1, part A and provide what your `get-num-cols` selector will return.

```
> (get-num-cols (get-board ____________________________________))

==>_____
```

Fill in the first blank below with your board from Exercise 1, part D, then fill the second blank in with a valid location on this board, and provide what your `get-piece-at` selector will return.

```
> (get-piece-at ____________________________________ ____)

==>_____
```

Putting it all together…

d) Now that you have a *board* constructor, the `make-position` constructor should be a cinch. Assume you have written `make-position` and, **using `make-board`**, create the position from exercise 1, part A.

```
> ____________________________________________________________________
```

Exercise 4: Representing the Move

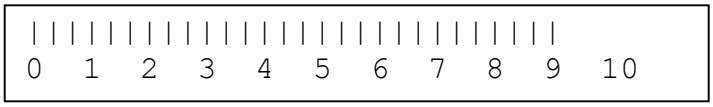
A *move* in Gamesman is a representation of how you might physically move a piece on the board. A move does not contain any information about the rest of the pieces on the board. Rather, it represents one move made by one player. Also, because you can tell whose turn it is just by looking at the position, the move does not contain information about which player is making the move or whether the move is good or bad (winning or losing). The representation should work for all types of moves, no matter if the move will result in the player `winning`, `losing`, or `tying`.

Lets look at a Gamesman example from *1,2,…,10*. A move in *1,2,…,10* is represented as a number (1 or 2). At most positions, the player can choose to add 1 or 2 pieces to the board, bringing the total closer to 10 (except at position 9 where the player can only choose to put down 1 piece). Here's an example:

```
Position: (l 7)
Whose Move?: Left
```

```
|||||||||||||||||||||
0  1  2  3  4  5  6  7  8  9  10
```

```
Moves: 1, 2

Position: (r 9)
Whose Move?: Right
Moves: 1
```



A move in *Tomorrow's-Tic-Tac-Toe* is also represented as a number, referring to the particular slot where a piece is to be placed (see the legend below). Here are a few examples of moves given the following position. (Note: There are no actual function calls being made below.)

```
Position:
(x --- --- ---)
Whose Move?: X-player
Moves: a1, a2, a3, b1, b2, b3, c1, c2, c3
```

```
  3  |  - - -
  2  |  - - -
  1  |  - - -
        a b c
```

```
Position:
(o -x- xoo --x)
Whose Move?: O-player
Moves: a1, c1, a3, b3
```

```
  3  |  - - x
  2  |  x o o
  1  |  - x -
        a b c
```

Now consider your own move representation for your game. They don't need to be numbers, like in *1,2,...,10*, unless it suits you to do so. Remember that the four games you choose from may have different kinds of moves than *Tomorrow's-Tic-Tac-Toe* and *1,2,...,10*. Instead of just placing new pieces on the board, some of these games involve a piece jumping from one location to the next. Thus, the move should include information about which piece is being moved, and where to. Pretend that in *Tomorrow's-Tic-Tac-Toe* you were allowed to slide one of your existing pieces around instead of putting a new piece on the board. What would it take to represent, for example, a slide from slot c2 to slot b1?

Once again, the way you choose to represent your moves is *entirely up to you*. **Gamesman** puts no restrictions on what form your moves must take, except that you are limited to sentences and words as before. It is also a good idea to make appropriate selectors for your move, even if they are as simple as a mapping to `first` or `butlast`. It is important that your program read well so that you don't get tangled in a web of Data Abstraction Violation (DAV) `first`s and `butfirst`s as you write your game. The time it takes to write a simple selector which will allow you to easily read your program is far less than the time you will spend debugging the one missing `butfirst` you forgot, not to mention the frustration you will save yourself.

a) In later weeks of this project, you will be asked to write a function called `generate-moves`. This `generate-moves` function takes one argument, a position, and returns a **list** (**not** a sentence or a word) of all the moves available for the player whose turn it is. For now, don't worry about how this is done. This is a more challenging function than others in this worksheet, but you will be able to tackle it easily using the abstractions you will create from above.

This exercise will simulate `generate-moves`. In the first blank below, please enter your board representation for Exercise 1 Part a. Then fill out the rest, using standard play options (don't apply any fancy rules, just a standard game as stated in the game sheets). You need not use all the spaces (below) for moves, each unique move should be written once *and only once*, and there's no particular order to the moves. You may not need all the blanks…

```
Position: _____________________________ (copy position from 1.a)

Whose Move?: _____________________________ (copy from 1.b)

Moves: __________, __________, __________, __________,
       __________, __________, __________, __________,
       __________, __________, __________, __________,
       __________, __________, __________, __________,
       __________, __________, __________, __________,
       __________, __________, __________, __________.
```

b) In the upcoming weeks, you will also be required to write `do-move`. This function takes two arguments, a *position* and a *move*, and returns the new position which results from performing that move on that position. Here's an example from *1,2,...,10*:

```
> (do-move '(1 7) 2) ;; add 2 to the pile so far
==> (r 9)
```

And here's an example from *Tomorrow's-Tic-Tac-Toe* …

```
 > (do-move '(x x-- x-- o-o) b3) ;; place an x on slot b3
==> (o xx- x-- o-o)
```

Notice that the calls to `do-move` change the element that determines whose move it is. In *1,2,...,10*, `1` changes to `r`; in *Tomorrow's-Tic-Tac-Toe*, `x` changes to `o`. Now, assume you've already written `do-move` for your game. Below, fill in the first blank with your position from Exercise 1 Part A. For the second blank, circle one of moves you created for Part A of this question (2.A) and write it in. Then provide the new position that would result from the `do-move` function call.

```
> (do-move _____________________________________________   ___)
==> _____________________________________
```

Exercise 4: Primitive Positions

While a game is being played, Gamesman needs to check each position to see if the game has ended. These end-game positions are known as *primitive positions*. A primitive position is a `win`, a `loss`, or a `tie` for the player whose turn it is. If a position is not primitive, then the game continues.

Gamesman determines whether or not a position is primitive by calling `primitive-position`, a function you will be required to write for your game. The function takes one

argument, a position, and returns the word `w`, `l`, or `t` if the position is primitive. (We use `w`, `l`, or `t` instead of the words `win`, `loss`, or `tie` for simplicity and to make the game tree more readable.) If the position is not primitive, then `primitive-position` returns `#f` for that position. Notice that some games have primitive positions resulting in `ties` (e.g., *Tomorrow's-Tic-Tac-Toe*); others only `wins` and `losses` (e.g., *1,2,...,10*). None of the four games *you* have to implement have ties, however.

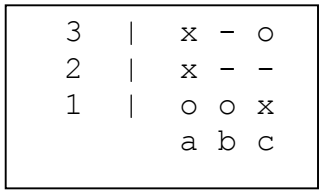
Here are a few sample calls to `primitive-position` from *1,2,...,10*:

```
(primitive-position (l 9))  ==> #f    ;; (game is not over)

(primitive-position (r 10)) ==> l     ;; (Right player loses)
```

Because the implementation of primitive positions is sometimes difficult to understand, let's step through a more detailed example in *Tomorrow's-Tic-Tac-Toe*.

We begin with this position: `(x x-o x-- oox)`

Notice that it is `X-player`'s turn. This game is obviously not over because neither player has three in a row, nor have all the spaces been occupied. Thus, a call to `primitive-position` would return the following.



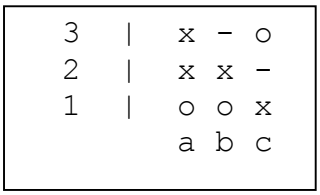

```
(primitive-position '(x x-o x-- oox)) ==> #f
```

Here are the available moves for the `X-player: b2, c2, b3`

It is obvious that if `X-player` chooses b2 as a move that he will win the game. Thus, after choosing this move, the next position is created by calling `do-move`:

```
> (do-move '(x oox x-- x-o) b2)
==> (o oox xx- x-o)
```

Now let's look at the resulting position:
`(o x-o xx- oox)`



Notice that it is now `O-player`'s turn, not `X-player`'s anymore. But now there are three in a row on the board, so that means that the game is over. And because it is currently `O-player`'s turn, this game ends in a loss for `O-player`. A call to `primitive-position` would return the following:

```
(primitive-position '(o oox xx- x-o)) => l
```

Remember, even though the `X-player` won, `primitive-position` returns an `l` (for `loss`) because it is currently `O-player`'s turn.

Assume `primitive-position` is already written for your game. You are required to fill in the function calls below with three **different** *positions* for your game. The first position must result in a `loss` for the White/Left player, the second must result in a `loss` for the Black/Right player, and the last position *must not be primitive*.

```
(primitive-position _____________________________________________)
=> l (lose for White/Right/O)

(primitive-position _____________________________________________)
=> l (lose for Black/Left/X)

(primitive-position _____________________________________________)
=> #f (not primitive)
```

There are many more primitive positions possible in your game. Start thinking about how you might write `primitive-position`. Perhaps you should consider using some of the functions you will have defined as part of prior checks-offs.

A few final notes…

1. **DO NOT WAIT UNTL THE LAST MINUTE TO START.** This is not a small undertaking. You will likely run into road blocks either in design or implementation. You will want to discuss various approaches with your lab assistants, TAs, and Dan.

2. Once you have written and individually tested all of your required procedures, you should first play your game using `test-loop.scm`. Next play your game **without** solving in `gamesman`. This will prevent wasting both your time and server computing resources. You will also be able to play and test much larger boards without solving because the computer can only "solve" relatively small boards. Although you will not be able to access the extra features of gamesman associated with a solved game (such as "safe" moves or win/loss prediction) they would be meaningless without a game that works properly. Solve and play **only after you have tested it without solving and you are sure it works**.

3. **HAVE FUN!** After all, it's only a game, and games are supposed to be fun.