Overview

This is the last assignment of the semester. Yippee! However do keep in mind that the project will not be graded face-to-face. Instead, you are to hand in a 10" x 13" envelope containing your submission. Guidelines have been specified on what you are to hand in and how you are to do it. (Refer the handout "General Information For Project")

Up to 20 points will be awarded for project correctness and adherence to specified turn-in procedures; up to 20 points (that's **half the grade**) will be awarded for displayed correctness (test cases), readability, and the general case you make that your program works correctly.

Readability includes comments, indenting, sequence of functions in your file, appropriate use of Scheme, and choice of names for functions and their inputs. Some readability guidelines: Functions that deal with the same kind of data structures should be together in your file, so that they are easy to find.

You do not need to write comments to explain the purpose of the functions that are described in the project description beginning on the next page. For other functions that you write, you should include comments that explain roughly how your functions work, unless it's obvious how it works at a glance. Each auxiliary function should be accompanied by a description of the function's purpose, a sample call, and the types and expected values for its inputs.

It's **very** important that you test your component functions in isolation, to provide evidence that they will work together correctly. It is critical that you print out the results of all these tests and save them in a file called `isolation.txt`. When the grader reads your listing of test cases, he/she should have **no doubt** that your program works as intended.

You should test the overall program `(gamesman)` too. Print out the listings for some sample games with different rules (more on that later). Save the output of your tests of the completed program in a file called `overall.txt`. Thus your named disk should have only three files on it: (one of `mkayles.scm, mtoads.scm, mpoison.scm, msnake.scm` or `mtactix.scm`), `overall.txt` And `isolation.txt`.

Do not under any circumstances turn in code that was not written by one of the members of your partnership. Sharing solutions with other groups on the project is the same as cheating on an exam.

The description of the program that you are to write begins on the next page. Good luck!

We're looking for a few good programmers…

There are many reasons to be excited about this project. The first is that it will be a whole lot of fun. The second is you get a chance to stretch your creativity! Several project requirements allow your team to design a component of the game of your choosing (rules & graphics). The third is (drum roll, please) "the SIGCSE Nifty Assignments" panel.

"Huh!?" you ask? This project has been deemed a "nifty" assignment and will be presented to the ACM's Technical Symposium on Computer Science Education (SIGCSE) conference to be held in late February / early March in Kentucky. Professors, instructors and teachers from around the world will be listening.

Dan and the staff will choose the best project for each of the five games and present them (with names and photos of the authors) to the gathered throngs at the conference. These students will be immortalized; this is one of the highest honors a CS3 student can receive.

The choice of the best project will primarily be based on the creativity (and number) of the group-chosen rules and the beauty of the graphics. These components are discussed in detail in the following page.

Background

Gamesman is a software library for solving and playing two-person, complete-information (no change or hidden information) board games started by Dan Garcia in the summer of 1991. This software works with game modules which describe the rules and winning / losing conditions of a particular game. The *Gamesman* system then *solves* the game (i.e., plays every game against itself and makes a table of the value at each position), and provides an interactive platform to play and analyze it. With this table at hand, *Gamesman* plays *perfectly*, and by doing so provides the ideal opponent for a player wishing to improve their skill.

The system is quite nice in that there is relatively little work to implement a game. The programmer just fills in a few required functions and then sits back while *Gamesman* does all the work solving it and controlling the play.

The idea to make this a CS3 final project has been brewing for some time. To achieve that goal, we brought in expert scheme programmers David Schultz and Greg Krimer, who have ported the code to Scheme for us. We've decided to implement much of the (often tricky) mathematics underneath each game. When you see the phrase "`hash`" and "`unhash`" in your code, feel free to ignore it unless you are truly curious.

Compulsory and User-chosen Rules

Another benefit of *Gamesman* is the ease with which a programmer can change the rules in a subtle way to create a game with a quite different strategy. Each of these new "rule options" is then presented to the user to tweak at their whim. For example, let's say we're playing the game "1,2,…,10" and we decide to let each player add 1, 2 or 3 (instead of only 1 or 2) on their move. This would be the game "1,2,3,…,10", and it might have a very different strategy from the original game…or it might be quite similar. Either way, the actual code that needed to be changed is quite minimal, and it allows us to put "a fresh face" on an old standard. Each of the five games listed below has a set of *compulsory* rules that each group must implement. In addition, you must come up with your own rule and implement it (called *user-chosen* rules), to create a completely new game. If you wish, you may implement more than one *user-chosen* rules – be as creative as possible!

Graphics

Very soon you will learn about graphics and fractals. You will use this knowledge to implement a graphical representation of your game's "board", or "position". We will provide you with `graphics-lib.scm`, a library of graphics routines you'll probably find useful. You are encouraged to be creative when drawing your board. Unfortunately, there is no graphical input, just output.

Problem

In lecture, we taught you the basics of combinatorial game theory and introduced you to seven two-person, complete information games:

- **1,2,…,10** [example, full source code `m1210.scm` provided]
- **Tic-tac-toe** [example, full source code `mttt.scm` provided]
- **Kayles** [template `mkayles.scm` provided]
- **TacTix** [template `mtactix.scm` provided]
- **Toads and Frogs** [template `mtoads.scm` provided]
- **Snake** [template `msnake.scm` provided]
- **Poison** [template `mpoision.scm` provided]

Your job, should you choose to accept it, is the following:

- Play **1,2,…,10** and **Tic-tac-toe** and look through their source code (check out `m1210.scm` first, it's the easiest) to figure out the basics of the control flow and how the game module interacts with `gamesman.scm`.
- Choose one of the latter five games: **Kayles**, **TacTix**, **Toads and Frogs**, **Snake** or **Poison**.
- Implement the game in Scheme by filling in several key procedures in the template Scheme file we'll give you (the hard mathematics has already been done!) This will take several weeks and should be started as soon as possible.
- Test your procedures thoroughly in isolation as well as within the *Gamesman* structure. (I.e., play the game quite a lot by running `(gamesman)` to make sure you handle all the hard cases).
- Implement the *compulsory* alternative rules so that you can effectively create several games for the cost of one.
- Test thoroughly.
- Choose and implement the *user-chosen* nifty rule(s).
- Test thoroughly.
- Design a cool graphical front-end for your game.
- Test, test, and test some more. Save your tests in `isolation.txt` and in `overall.txt`.
- Document your code, tie off any loose strings, and submit it.

Definitions

This section serves to clarify some of the terms related to game theory that are useful to understand. When appropriate, examples from the games Tic-Tac-Toe and "1,2,...,4"[1] will be supplied to illustrate the terms.

Position

Another word for this is a "board configuration" – it is a snapshot of the board with pieces on it and a turn designation. Every game has positions, regardless of whether or not it is played on a physical board. Figure 1 shows a sample position for Tic-Tac-Toe. Note that the outcome of the game would be quite different if it were O's turn. Impartial games do not need to have a turn designation (see the definition of impartial games below). E.g., for the abstract game "1,2,...10", the integers 1 through 10 are positions. As a matter of fact, if we make the assumption that X always starts, we can determine whose turn it is by counting the number of Xs and Os. When that number is even, it's X's turn, otherwise it's O's turn. Thus, when we are encoding the position in scheme, we do not need to store the player whose turn it is.

Figure 1: A Tic-Tac-Toe position

Slot

A slot is a coordinate on a board, which in the 2-D case would be an (X, Y) pair. A board, for some games, is a 2-D collection of slots. Keep in mind that abstract games such as "1,2,...4" do not have clear definitions of slots. In the game Tic-Tac-Toe, there are 9 slots, numbered in Figure 2.

Figure 2: The 9 Tic-Tac-Toe slots

Move

"Move only if there is a clear advantage to be gained"
– Sun Tzu
The Art of War [Tzu83, p. 83]

Moves are the action a player performs on his/her turn to change the board's configuration. In Tic-Tac-Toe, a move is the action of placing an X or O on an empty slot. Most games' rules dictate that the available moves are a function of position; in the game "1,2,...,10", however, there are exactly two possible moves for *every* position, 1 and 2. Most games dictate that the players alternate turns every other move.

Value, or Outcome

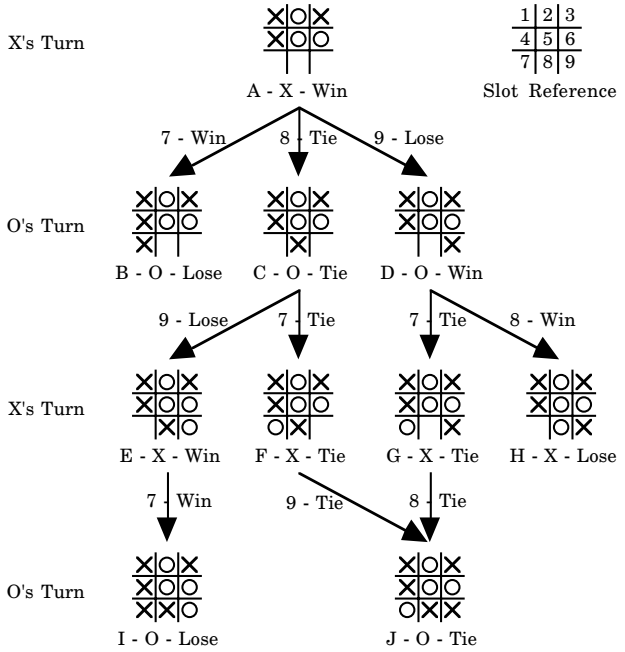

Figure 3: A branch in a Tic-Tac-Toe game-tree

We have simplified the meaning of the value of a game from that usually discussed in combinatorial game theory circles. Every position has a value, which we will consider to be one of { Win, Lose or Tie } for the player whose turn it is to move. This can also be thought as the outcome of the game if played by perfect opponents. This means that if the game was played between perfect opponents, the player whose turn it was would *always* either win, lose or tie. Any move that leads to a winning position for the other player is a *losing* move, and consequently any move that leads to a losing position for the other player is a *winning* move. A tie move is one that leads to a tie position for the other player. If the first, or initial position in a game has value V, then the *game* is said to have value V. For example, if Tic-Tac-Toe began with position A in Figure 3 rather than a blank board, then Tic-Tac-Toe would be a winning game, since A is a winning position.

Figure 3 contains a very detailed Tic-Tac-Toe game tree to help us understand these terms. Note that under every position is a string that

[1] A variation on the popular game, Nim, described in Appendix B.

represents the position letter, whose turn it is and what the value of that position is. For example, the root position contains "A - X - Win", which means that it is position A, X's turn and a win for X. The arrows have a number and a letter beside them that represent which slot was chosen and what the value of that move was. For example, the upper-left-most arrow contains "7 - Win", which means that player X chose slot 7, and it was a winning move for player X.

Win

"For nothing can seem foul to those that win."
– Shakespeare
Henry IV, Part I, Act V, Scene 1

In some references this is referred to as an "N" position, which means the Next player can win. This value is recursively defined by the following rule: *A winning position is one in which there exists a losing child*. This is best illustrated by position A in Figure 3, which has a winning (D), losing (B), and tieing (C) child, and is considered a winning position due to the existence of B. The move that leads to the losing child, slot 7, is the winning move. The following positions are all winning in the above game-tree: A, D and E.

Just because a player has a winning position doesn't necessarily mean the player *will* win, simply that the player *can* win. In position A above, a winning position, if X chooses the lone losing move to slot 9, X can lose. Sometimes a win is inevitable, since all the children are losing positions, and in these rare cases a winning position indicates that the player *will* win. Position E has one lone slot for X to choose which forces the win. X has no option but to choose slot 7 and win the game. It is important to remember that a winning position in general means that the *potential* for winning against a perfect opponent exists.

Lose

"Dr. Pulaski: To feel the thrill of victory, there has to be the possibility of failure. Where's the victory in winning a battle you can't possibly lose?
Data: Are you suggesting there's some value in losing?
Dr. Pulaski: Yes, yes, that's the great teacher. We humans learn more often from a failure or a mistake than from an easy success.
– Dr. Pulaski and Lieutenant Commander Data
in Star Trek : The Next Generation's *Elementary, My Dear Data*

Similar to a winning position, a losing position is often called "P", which means the Previous player can win. Said another way, it means that the player whose turn it is *will* lose against a perfect opponent. This value is also recursively defined, but by a different rule: *A losing position is one in which there does NOT exist a losing or tieing child*. This means that the children of losing positions are either all winning or it is primitive (and has no children). The losing positions from Figure 3 (B, I and N) fulfill the latter case and are all primitive losing positions. We will describe primitive positions in a moment.

If a player has a losing position and is playing against a perfect opponent, the player has already lost the game and might as well concede. This is because a perfect opponent will continue to make winning moves until either it has reached a primitive winning position or the other player is left with a primitive losing position. However, if the opponent is imperfect, then a non-primitive losing position does not guarantee a loss, just the *potential* for losing.

Tie

"I wish it could have been a tie"
– Amanda Bonner (Katherine Hepburn),
after defeating her husband in *Adam's Rib*.

A tie position is recursively defined as: *A tying position is one in which there does not exist a losing child, but there does exist a tie child.* Whether or not there are any winning children is irrelevant, as it does not affect the value. Position C in Figure 3 above is a perfect example of a tying position, since it has no losing child but *does* have a tie child, which is position F. A player with a tie position can either tie[2] or lose against a perfect opponent. Against an imperfect opponent, it is possible to either tie, lose or win. Positions C, F, G and J are tie positions, yet only J is a primitive tie. As mentioned before, a tie move is one that leads to a tie position. If there are no tying terminating criteria (see below), there can never be a primitive tie position, and by induction, no non-primitive tie positions.

Primitive positions and terminating criteria

Primitive positions are the leaves in the game tree and are the positions that fulfill the *terminating criteria* for the game. These criteria are what prevent the game from being infinite, as they force the game to end at some point. In Tic-Tac-Toe, there are two terminating criteria. The first is that the other player has just achieved three-in-a-row of his/her piece, which means that the position is a losing primitive position. The second is that a position has all 9 slots filled, in which case it is a tying primitive position. Note that for most games the order that these are checked is crucial, as position I fulfills both, and would be incorrectly labeled a tying position if the rules were reversed. Positions that are not primitive are either called *non-primitive* or *recursively-defined*, due to the definitions highlighted above. In Figure 3, the primitive positions are B, I, J and H.

'Safe', or value-equivalent moves.

Safe moves are moves that lead to children of equal value as the original position. E.g., all winning moves from a winning position are safe. All tying moves from a tie position are safe. Lastly, all losing moves from a losing position are safe (and futile!). All five moves available to X

[2] Since this is the best that can be hoped for, we suggest that players consider it a "win" to tie a perfect opponent given a tie game, such as TicTacToe.

(slots 2, 4, 5, 7 and 8 in Figure 4) are winning since all five lead to losing positions. These moves are *safe* and any may be chosen without risk to the outcome. This is true even if though 2 leads to an immediate primitive losing position (for O) and 4, 5, 7 and 8 lead to non-primitive losing positions (for O).

Figure 4: A winning Tic-Tac-Toe position with three safe winning moves 2, 4, 5, 7 & 8

Perfect opponents

"He wins his battles by making no mistakes. Making no mistakes is what establishes the certainty of victory, for it means conquering an enemy that is already defeated."
– Sun Tzu
The Art of War [Tzu83, p. 20]

Perfect opponents are opponents who *always* choose winning moves given winning positions and *always* choose tying moves given tying positions. It doesn't matter what perfect opponents choose given losing positions, since if they are primitive there are no moves available and the game is over (and if they are non-primitive all moves are value-equivalent 'safe' losing ones). It is tempting to define perfect opponents as computers and imperfect opponents as humans, but that would incorrectly label bad programs and very knowledgeable humans.

The misère game vs. the standard game

Every game has terminating criteria that constitute the rules of the game. These are called, for convenience, the *standard* rules, and must include winning or losing conditions. Every single game has a *misère* game, which is the game with the words "losing" and "winning" swapped into the terminating criteria for the standard game. For example, whereas the standard game in Tic-Tac-Toe is be explained as: "A player *wins* if he/she achieves three-in-a-row first", the *misère* game is explained as: "A player *loses* if he/she achieves three-in-a-row first".

Interestingly enough, sometimes perfect strategies for the standard game and the misère game differ by only the primitive positions and perhaps a few others. Conversely, some games have completely different strategies for the two games. Everyone must implement both standard and misère rules for the game they choose.

What you must implement

There are only five key procedures you will need to implement for each game: `do-move`, `primitive`, `print-position`, `print-help` and `generate-moves`. These are described below with Degree of Difficulty (DoD listed, from 1-10) and when they are due:

`print-position` (DoD 2) – Due at Checkoff 1

```
;;; PRINT-POSITION takes a position and prints it pretty as well
;;; as calls the graphics routines to render the position.
```

`print-help` (DoD 2) – Due at Checkoff 1

```
;;; PRINT-HELP takes a position and prints the rules and objective.
```

`do-move` (DoD 4) – Due at Checkoff 1

```
;;; DO-MOVE takes a position POS and a move MOVE and returns the position
;;; that results from making that move from that position.
```

`primitive` (DoD 6) – Due at Checkoff 2

```
;;; PRIMITIVE takes a POSITION and returns WIN, LOSE or TIE if the game is over.
;;; Otherwise, it returns UNDECIDED.
```

`generate-moves` (DoD 10) – Due at Checkoff 2

```
;;; GENERATE-MOVES takes a position and returns all the possible moves
;;; from that position. In 1,2,...,10, you technically can't move two
;;; spaces from the position (- board-size 1), but we pretend that you
;;; can by adding an imaginary spot to the board!
```

Running *Gamesman* from MacGambit

If you are using MacGambit, you need to have four files on your computer (downloadable from the main Gamesman site below): `gamesman.scm`, `graphics-lib.scm`, `m1210.scm` and `mttt.scm`. First load `graphics-lib.scm` and then load the module (which loads `gamesman.scm` and `graphics-lib.scm`). Then type `(gamesman)` to start the game.

Running *Gamesman* from Dr. Scheme (any platform)

If you are using Dr. Scheme, you need to first download and install the `graphics-teachpack` from the CS3 page. Then you'll need the same four files on your computer as described above. Open `m1210.scm` or `mttt.scm` and click "Execute". Then type `(gamesman)` to start the game.

"Shall We Play a Game" main www site

This document should be enough to get you started through Checkoff 0. The rest of the details you'll need for your project, FAQ (frequently asked questions), updates, etc., can be found at the following www page:

`http://www-inst.eecs.berkeley.edu/~cs3/gamesman/`

If you have questions or bugs, please email:

`cs3-gamesman@nim.cs.berkeley.edu`

…which will be read by the main project architects who will answer all questions and place them on the FAQ page on the gamesman site above.

Enjoy! -- Dan, Greg and Dave [and the rest of the CS3 staff]