CS 312 Problem Set 6: Abalone

Assigned: April 11, 2008

Final submission due: May 1, 2008, 11:59 PM
Design meetings: April 16–20, 2008

1 Introduction

In the previous assignment (Problem Set 5), you developed an interpreter for a concurrent programming language. This part will allow you to put that language to good use: you will develop a version of the game *Abalone*. In this game, two players each attempt to move their pieces to push their opponent's pieces off the board. In your version of the game, each game piece will be controlled by a separate CL thread.

You will implement the mechanics for this game in SML, as well as the code for a game player, in CL. Your evaluator from Problem Set 5 will be used to run the programs controlling the two teams. You should be able to reuse your Problem Set 5 code without changes, except perhaps to fix bugs. As in Problem Set 4, we also have provided some graphical support that you can use to display the game. Source code for getting started on this project is available in CMS. You will keep the same partner you had for the previous assignment; consult Professor Myers if this is truly problematic.

There are few constraints on how you implement this project. This does not mean you can abandon what you have learned about abstraction, style and modularity; rather, this is an opportunity to demonstrate all three in the creation of elegant code.

You start by carefully designing your system, and presenting this design at a *design meeting* partway through the assignment where you will meet with a course staff member to discuss your design. You are required to submit a printed copy of the signatures for each of the modules included in your design at the design meeting. Part of your score will be based on the design you present at this meeting.

1.1 Reading this writeup

This writeup refers to constants written in the code font, such as WINNING_SCORE. The values of these constants are defined in the source file definitions.sml. You should briefly familiarize yourself with those constants to fully understand this writeup.

1.2 Updates to Problem Set

Any updates other than minor fixes will be recorded here.

- Scoring has been slightly modified: a team now also gets a point when an opposing piece goes away due to it's controlling thread terminating, in addition to getting points for pushing opposing pieces off the board

1.3 Point Breakdown

- Design meeting - 5 pts
- World - 45 pts
- CL team (AI) - 15 pts
- Documentation and design - 10 pts
- Barrier abstraction - 10 pts
- Written problem - 15 pts

2 Game Rules

Abalone is a two-player strategy board game that involves positioning and coordinating pieces to push opposing pieces off the board. The official rules for Abalone are available at http://uk.abalonegames.com/rules/basic_rules/official_rules.html. A number of helpful diagrams and examples can be found at http://en.wikipedia.org/wiki/Abalone_(board_game). However, we will be using a modified version of the rules, so you should rely on this writeup instead.

2.1 Scoring and Winning

There are two teams, the black team and the white team, named after the color of their pieces. Teams score one point for each opposing piece that leaves the board, as a result of either pushing or the controlling thread terminating. A team wins the game when they reach WINNING_SCORE points. If no team reaches WINNING_SCORE points by the time GAME_LENGTH cycles have passed, the team with the highest score wins. If both teams have an equal score, the team with more pieces remaining wins. If both teams have an equal number of pieces remaining, the game is a draw.

2.2 Board

Abalone is played on a hexagonal board with 61 hexes, arranged in a hexagon with five hexes to a side. Each hex is addressed by its row letter (A–I) and its diagonal number (-4 to 4). Lettering for rows starts with the bottom row as A, and proceeds upward. Numbering for diagonals starts with the lower-leftmost diagonal as -4, and proceeds to the right, so the bottom left hex as A-4. Coordinates will generally be written as integer pairs (r, d), with each letter A–I assigned a numeric value, starting with A equal to -4. For example, we could write hex A2 as $(-4, 2)$. For example, on the board pictured in Figure 1, the piece marked X is on hex G0 $= (2, 0)$, the piece marked Y is on hex C-3 $= (-2, -3)$, and the piece marked Z is on hex H4 $= (3, 4)$.

Note that both rows and diagonals vary in length. Length starts at five for row A and diagonal -4, increases by one for each new row or diagonal until reaching a maximum of nine for row E and diagonal 0, and then decreases by one for each new row or diagonal until reaching five again for row I and diagonal 4.

The coordinate system is more regular than it might appear at first glance. The legal positions on the board are exactly those (r, d) where $|r| \leq 4$, $|d| \leq 4$, and $|r - d| \leq 4$. It is also possible to translate and rotate points. In particularly, a $60°$ right rotation transforms (r, d) into $(r - d, r)$, while a $60°$ left rotation transforms (r, d) into $(d, d - r)$.

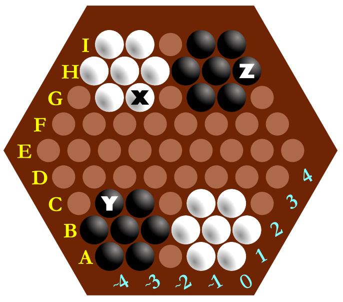

Figure 1: The board and coordinate system

2.3 Directions

Each hex is adjacent to up to six other hexes. This gives each piece up to six possible directions to move. The coordinates of the hexes adjacent to a given hex (r, d), and the direction to move to reach them, are as follows:

- $(r + 1, d)$ (up-left, UL)
- $(r - 1, d)$ (down-right, DR)
- $(r, d + 1)$ (right, R)
- $(r, d - 1)$ (left, L)
- $(r + 1, d + 1)$ (up-right, UR)
- $(r - 1, d - 1)$ (down-left, DL)

Note that the hexes $(r + 1, d - 1)$ and $(r - 1, d + 1)$ are *not* adjacent to (r, d), and that just "up" and "down" are *not* valid directions.

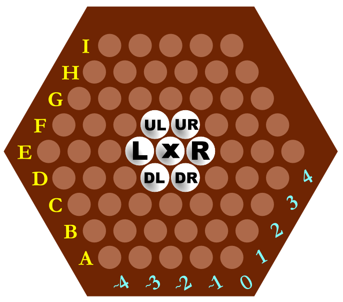

Figure 2: The adjacent squares for the piece marked X at $(0, 0)$, marked with their direction from that piece

2.4 Pieces

In this version of Abalone, each piece is actually a robot controlled by its own CL thread. We refer to these pieces as bots. Pieces are moved on the board when the bot performs an appropriate *action* invoked using the CL do expression. Bots can also determine the position of pieces on the board and other game information by making use of actions.

Bots can communicate with each other using the CL shared memory. The CL concurrency mechanisms may be useful in making this work.

Pieces that are pushed off the board are considered dead, and the the associated thread is terminated. Conversely, if a thread terminates, the piece it represents must be removed from the board.

2.5 Teams and initial positioning

Each team begins with a single piece (at position BLACK_START for the black team, and position WHITE_START for the white team). Each team may, before its first move occurs, spawn additional pieces, until it has reached MAX_PIECES pieces.

2.6 Spawning New Pieces

A new piece is created whenever a new process is successfully spawned. Newly created pieces are placed in one of the empty hexes adjacent to the bot doing the spawning, chosen randomly. New processes may be spawned by a team only under the following conditions:

1. At least one of hexes adjacent to the spawning bot is empty.
2. It is before the end of the team's first turn.
3. The team controls less than MAX_PIECES pieces.

If these conditions are not satisfied, the spawn fails. The new process created by the spawning controls the new piece.

2.7 Scheduling

How much each team is allowed to evaluate is important to the fairness of the game. You are going to use the evaluator from Problem Set 5 to evaluate the CL code for the bots. In each *clock cycle*, every bot of each team is stepped exactly once. The world is then notified that a cycle has ended to that it can perform any necessary updates.

Every CYCLES_PER_TURN cycles, moves are processed for one of the two teams, starting with the black team and alternating after that. A turn which has the white team's moves processed at the end is referred to as a white turn, and similarly for black turns.

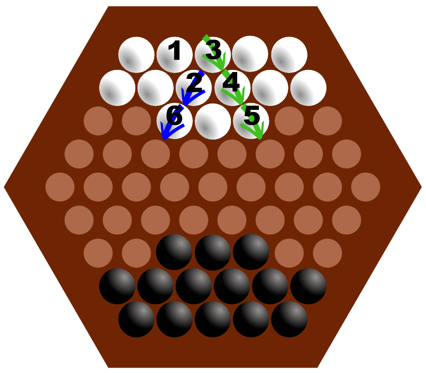

Figure 3: Chain selection example. Pieces 2 and 6 are selected as the first chain. Pieces 3, 4, and 5 are selected as the second.

2.8 Movement

A bot moves by performing an A_COMMITMOVE action. The movement does not occur immediately; it occurs on its team's next turn to move. The bot continues to execute after performing the action. At regular intervals during execution, the pieces of one team or the other (alternately) move. Bots that have committed a move get a chance to move when their team's turn comes up; other bots do not. Depending on how many bots commit moves and how many of those moves are legal, a team may have anywhere from all to none of its pieces move on a given turn. No bot ever moves twice in a turn, and no moves ever happen except at the specified intervals.

2.9 Movement Order and Chains

Since a team may have multiple moves committed, and the result of the moves may be different depending on what order they are executed in, we must define an ordering for move execution. We define such an ordering by going by lowest PID first, and grouping pieces into chains, as described in the following sections.

A *chain* is defined as a group of one or more pieces that are adjacent to each other in a straight line, and have uncommitted, unperformed moves all in the same direction. If the chain contains more than one piece, the direction they are attempting to move in must be parallel to the line. At the end of each turn, all chains with committed moves from the appropriate team will attempt to move.

2.10 Chain Selection

During move processing for a given team, if no moves have been committed by the team, nothing moves, and the game proceeds to the end of move resolution. Otherwise, at least one piece has a committed move, and we must select a chain to move first (since what order chains move can change the result). The chain that will move is selected by determining the lowest PID piece with a committed move, and selecting the largest legal chain that includes that piece.

For a chain to be legal, all the pieces in the chain must be located in a straight, contiguous line. They must all have committed, still unperformed moves along the direction of the line.

If the move of the chain is legal as described below, the pieces in the chain are moved in the common direction committed by all the pieces. Then, regardless of whether the move was legal or not, the committed moves for all the pieces in the chain are reset, and this process repeats with the next bot with an uncommitted move, if any. Note that while this gives every piece a chance to move, it does not mean that every piece *will* move, as some pieces may find that their committed move is illegal. This can happen even if the move appeared to be legal when it was committed.

2.11 Move Legality

Next, it is determined if the move is legal. A chain can push up to two of the opponent's pieces as long as the chain contains more pieces than it is pushing. To be more precise, call the three hexes in front of the chain H_1, H_2, and H_3. A move is legal if and only if:

1. H_1 is empty; or
2. H_1 contains an enemy piece, H_2 is empty or off the board, and the chain is of length at least 2; or
3. H_1 contains an enemy piece, H_2 contains an enemy piece, H_3 is empty or off the board, and the chain is of length at least 3

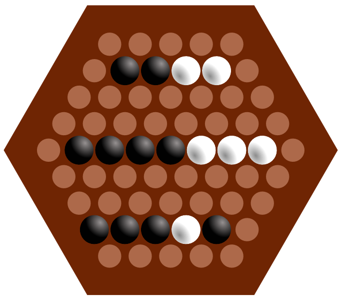
Figure 4: No legal move for black results in a push

2.12 Move resolution

If a move is legal, all pieces in the moving chain are moved one hex in the direction of movement, as are all enemy pieces directly in front of the chain (in the direction of movement). This movement of enemy pieces is referred to as "a push" or "pushing". If a push would cause an enemy piece to go off the board, that piece is eliminated and considered dead, and the thread controlling it is terminated. Pushing an enemy piece off in this manner increases the score of the moving team by one.

At the end of move resolution, the committed moves for all pieces on the processed team should be nothing. At this time, all processes that are waiting for the next turn should be woken, and all processes waiting for the next turn for their team that are *not* on the team which just moved should be woken. See the specification for actions for more details on the wait actions.

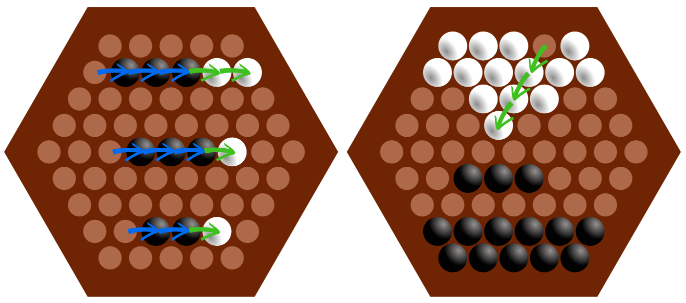
Figure 5: Various legal moves

2.13 Move Resolution Example

For example, consider the following board during chain selection for white, with the PID of each piece indicated by the number and the current committed move of that piece signified by an arrow (with no arrow meaning no committed). Piece 1 does not have a committed move, so piece 2 (the piece with the lowest PID that has a committed move) is selected as the start of the chain, and the largest chain that can be made using it consists of itself and piece 6, so the full chain selected is piece 2 and piece 6. The move is legal, so that chain is moved, and the committed moves for those pieces are set to nothing.

As there are still more pieces with committed on the white team, we now repeat the chain selection process. Piece 3 is now the lowest PID piece with a committed move, so it is selected as the chain start. Its chain is determined to be 3-4-5, so that chain is moved, and all the committed move for pieces 3, 4, and 5 is set to nothing. Since now all white pieces have no committed move, move resolution ends.

3 Actions

There are a variety of actions that the bots can perform via the do command, as described below. Each action returns a value. The effects of an action are visible to the world and the other bots immediately after the action is performed.

3.1 Action Quick Reference

A conceptual explanation of the available actions follows. Actions are specified in full detail below.

A_MYSTAT	Returns the position and direction of the piece
A_TEAMSTAT	Returns piece count, score, and number of spawns left for both teams
A_GAMESTAT	Returns the current turn number and the number of cycles until the bot's team's moves are processed
A_INSPECT	Returns the contents of a specified hex
A_LOOK	Returns an array of the contents of all hexes in a straight line from the piece
A_SCAN	Returns an array of the contents of all hexes adjacent to the piece
A_COMMITMOVE	Commits a move for the piece
A_WAITTURN	Waits until the start of the next turn
A_WAITMYTURN	Waits until after the next enemy turn
A_TALK	Outputs a string to the chat area

3.2 Action Specification

The figures on the following pages describe the possible actions in greater detail. Recall from PS5 that $(v_0, ..., v_{n-1})$ represents an array literal, which will evaluate to an array with the ith index for $i = 0, ..., n-1$ equal to v_i, and all other indicies equal to zero. Also, in these specifications, we mention various constants, which are defined in definitions.sml and constants.ch. Here are some of the constants that the actions refer to:

Object constants	T_EMPTY	An empty hex
	T_PIECE	A piece
	T_OPIECE	An enemy piece
	T_OFFBOARD	A hex off the board

Command	Args	Effects	Returns	
A_MYSTAT	None	None	$((r,d),dir)$	where (r,d) is the bot's position and dir is the bot's direction
A_TEAMSTAT	None	None	(pc,s,ns,pco,so,nso)	where pc is the piece count of the bot's team, s is the score of the bot's team, ns is the number of spawns remaining for the bot's team, and pco,so,and nso are the corresponding quantities for the opponent
A_GAMESTAT	None	None	(t,ttm)	where t is the current turn number and ttm is the time until the bot's team's next move
A_INSPECT	(r,d)	None	o	where o is the constant representing the object at position (r,d)
A_LOOK	dir	None	l	where l is a array of constants representing objects between the bot and the edge of the board in the given direction, with the closest object first, terminated by the constant representing an offboard hex
A_SCAN	None	None	(r,ur,ul,l,dl,dr)	where r is the constant representing the object in the hex to the right of the bot, ur is the upper right, etc.
A_COMMITMOVE	dir	Changes the committed move for the bot to dir	0	
A_WAITTURN	None	Blocks until the start of the next turn	0	
A_WAITMYTURN	None	Blocks until the cycle after the opponent's next move resolution	0	
A_TALK	s	Outputs the string s to the chat area	0	This action has been partially implemented for you for debugging purposes

4 CL Features and Interpreter Updates

We make note of some CL features that you may not have used much in PS5, but will be very useful for testing the game and implementing your bot. Further, to aid in your design we have added a few useful features to CL, none of which should require any modifications to your current interpreter. We have also made some slight modifications to the structure of the interpreter, to aid in implementing some aspects of the game.

4.1 Recursive Functions

The support for recursive function already present in CL will be extremely helpful for this assignment. To review, you can define recursive functions by using the keyword rec, as in the example below:

```
let fact =
  rec f in
    fn n => if n=0 then 1 else n * f(n-1)
in fact 3
```

4.2 Includes

You may wish to write code that multiple CL programs can use. You can do this using the #include command. The argument is the name of the file to include. Keep in mind that the path for the file should be relative to the directory from which you run SML, not the directory in which the CL file is located. If you execute SML from the project directory and want to include in a unit the constants.ch file that we have written, which is in the cl directory, you would use:

```
#include cl/constants.ch
```

When this line is read, the file cl/constants.ch is automatically loaded and its contents replace the #include line. You will most likely use #include to declare a bunch of commonly used functions. For instance, you might include a file called functions.ch with #include functions.ch, which contains

```
let trymove = fn x => if x = 0 then (do x)
                      else trymove (x - 1)
in let calcpos = fn x => fn y => fn z => ...
in
```

The file being included should end with in so that any code following the #include declaration is treated as the body of the let.

4.3 Array Library

We have provided a useful library for manipulating arrays, located in the file cl/arrays.ch. The functions in the array library are similar to those in SML, as shown below. The array library defines the following functions, all of which are implemented using the "rec" construct. All of these functions consider an array entry of zero to represent the end of the array.

foldl f acc l s	Fold the function f over the array l with the initial accumulator acc, starting with the index 0 and ending with the sth element.
foldr f acc l s	Fold the function f over the array l with the initial accumulator acc, starting with the sth element and proceeding to index 0.
map f l s	Map over the first s elements in list l with function f to return a new array.
append l1 l2 s1 s2	Puts the first s2 elements of l2 into l1, replacing the elements after the first s1 elements of l1

4.4 Other Libraries

An abstraction for representing and manipulating booleans is provided in booleans.ch:

true	Boolean constant true
false	Boolean constant false
and	A curried function that performs the logical conjunction of its arguments
or	A curried function that performs the logical disjunction of its arguments
not	Logical negation
xor	A curried function that performs the logical exclusive disjunction of its arguments

4.5 Interpreter Updates

We have updated the interpreter in the following ways. You will find these updates helpful for implementing certain actions. Note also that both world.sig and world.sml are completely different.

- We have added a BlockedByWorld constructor to the process_status datatype in Concurrency
- We have changed stepAndUpdate in Evaluation to account for BlockedByWorld
- We have exposed the releaseAndWake function in Evaluation, so that the game loop can use it

5 GUI

To display your game, we have provided a program that can graphically present a hexagonal board, with an area for score for each team, and the current turn number. There are interfaces for updating all the numerical fields, and for setting hexes to display either nothing or a piece trying to move in a direction.

The GUI must be updated when the game state changes, which includes changes in score, turn number, or piece positioning or direction. Note that as pieces are displayed with their direction, this requires an update after a piece commits a move. GUI updates do not need to be instantaneous, but must be fairly prompt.

A GUI command is just a string, starting with the command name, followed by arguments depending on the particular command. Commands can be sent to the GUI using the NetGraphics.report function. The GUI commands are as follows:

Command	Result
bsay ⟨string⟩	has the black team saying something (e.g., `bsay I'm the black team`)
wsay ⟨string⟩	has the white team saying something (e.g., `wsay I'm the white team`)
bscore ⟨n⟩	sets the black team's score to be n
wscore ⟨n⟩	sets the white team's score to be n
turns ⟨n⟩	sets the turn number to be n
set ⟨pos⟩ e	sets the hex given by pos to be empty
set ⟨pos⟩ ⟨w|b⟩⟨d⟩	sets the hex given by pos to be a (white|black) piece facing in direction d

For specifying positions, the GUI understands the row as a letter from A to I, and the diagonal as a number from ˜4 to 4. Either ˜ or - will work for negatives.

For specifying directions, the GUI understands integers from 0 to 6, with zero representing no direction and 1 to 6 representing the other directions, start with 1 representing right and going counterclockwise.

Number	Direction
0	No direction
1	Right
2	Up Right
3	Up Left
4	Left
5	Down Left
6	Down Right

For example, to set the hex at position $(B, -2)$ to a white piece facing left, you would send the command `set B~2 w4`. Note that there is no space between the `B` and the `~2`, and no space between the `w` and the `4`.

6 Your tasks

There are several parts to the implementation of this project. Make sure you spend time thinking about each part before starting. Start on this project *early*. There are many things you will have to take into consideration when designing the code for each section.

6.1 CL interpreter

For the game to work, the CL interpreter must be correct. For the game to work well, the CL interpreter must be reasonably efficient. We are not asking you to do any new implementation work on the CL interpreter, but you are expected to fix any bugs in the interpreter that you submitted for Project I, and make it run at a reasonable speed.

We have added new functions to some of the interpreter files to ease implementation of certain aspects of the game. For the updated interpreter files that are included in the PS6 download, you should merge any changes you made into the files. For the other interpreter files, you should simply copy over your files from PS5.

6.2 Designing the world

Your first task is to create a design for your Abalone implementation and meet with the course staff to review it. Your second task is to implement the *Abalone* world in the files `world/world.sml` and `world/game.sml`, and any files you choose to add. Note that you should add files only to the `world` and `cl` directories. You must implement the actions listed in Section 3. You must also make sure that the actions bots take are rendered in the graphic display using the interface detailed in Section 5. You can use the sample bot program we provide to test your world, but for full testing coverage you will need to write your own tests.

6.3 Designing a team

Your third task is to design a CL team in a file `cl/team.cl`. To receive any credit, your team must be able to consistently beat the team provided by the course staff, which is a very weak team. You will be graded on your bot's general strategy and how it does against a number of test bots, including the one provided.

6.4 Documentation

You should submit a design overview document for this project, just like the ones for the previous assignments. Since this project is both large and quite open-ended regarding the way one may choose to implement it, documentation becomes even more important. Your design overview should probably be as long or longer than your design overviews for the previous assignments.

6.5 Things to keep in mind

Here are some issues to keep in mind when designing and implementing the world:

- **Think carefully about how to break up your program into loosely coupled modules.** The program will be complex and difficult to debug unless you can develop modules that encapsulate important aspects of the game. Design the interfaces to these modules carefully so that you can work effectively with your partner and can do unit testing of the modules as you implement.
- **Make sure that what is going on in the world matches what is going on in the graphics.** Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember, it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the world is acting properly. It would behoove you to maintain some sort of invariant between the status of the world and the status of the graphics.
- **Problems in the world might actually be problems with the bots.** If you are using your own bots to test the actions and something seems wrong, the bots could just as easily be at fault.
- **Implement and test the actions one at a time.** Don't try to implement all of the actions and test them with one single team. Start with easier actions and work up to the harder ones. For example, start with a simple action like `mystat`.

There are also many different strategies for building a good team. Consider, for instance, that your bots can communicate and share memory that the opposing team cannot access. Use it to your advantage to coordinate your maneuvers.

6.6 Design meeting

For this assignment, there will be a *design meeting* partway through the assignment. Each group will use CMS to sign up for a meeting, which will take place between Wednesday, April 16 and Sunday, April 20. If you are unable to sign up for any of the available time slots on CMS, contact the course staff, and we will try to acommadate you.

At the meeting, you are expected to explain the design of your system, give a brief description of the design of your CL bots, and hand in a printed copy of the signatures for each of the modules in your design. In designing module interfaces, think about what functionality needs to go into each module, how the interfaces can be made as simple and narrow as possible, and what information needs to be kept track of by each module. Everyone in the group should be prepared to discuss the design and explain why the module signatures are the way they are. We will give you feedback on your design.

We strongly encourage that you come discuss your design with the course staff during consulting and office hours, both before and after the meetings.

6.7 Final submission

You will submit:

1. A zip file of all files in your project directory, including those you did not edit. We should be able to unzip this and run CM.make("sources.cm") to compile your code (i.e., you should include your `sources.cm` file). This should include:

   - your world implementation
   - your bot, with the bot named team.cl in the cl directory along with all the custom libraries it uses (if any)

   It is very important that you organize your files in this manner, as it greatly simplifies grading.

2. Your documentation file, in .txt, .pdf, or .doc format.

Although you will submit the entire project directory, you should only add new files to the world and cl folders; the other folder must remain unchanged. If you add new sml or sig files, be sure to modify sources.cm to include them. Note again that we expect to be able to unzip your submission and run CM.make("sources.cm") in the newly created directory to compile your code without errors or warnings. **Submissions that do not meet this criterion will be docked points.**

7 Tournament

Sometime during finals period, most likely on Tuesday, May 6, we will hold a competition for students who wish to compete. Each group may submit a CL team that will play against other students' teams. Details on the tournament time, location, and the submission procedure will be available later.

8 Provided source code

Many files are provided for this assignment. Most of them you will not need to edit at all. In fact, other than merging your changes from PS5 into the updated interpreter files in the `eval` directory, you should only edit and/or create new files in the `world` and `cl` directories. Here is a list of all the files and their contents.

`gfx/*`	Graphics files for the GUI
`eval/*`	Updated versions of some interpreter files (you should merge your changes from PS5 as necessary)
`world/definitions.sml`	Definitions of a wide variety of constants
`world/world.sig`	Signature file for handling actions and world state
`world/world.sml`	Functions for handling actions and world state
`world/game.sig`	Signature file for the functions contained in the game
`world/game.sml`	Handles the game state
`world/loop.sml`	Starts and continues the main game loop
`world/*`	Other utility files for the world
`net/*.sml`	Network SML code for communicating with the GUI
`gui/*`	The GUI files
`cl/*.ch`	CL libraries
`cl/simple_bot.cl`	Sample team program that you need to beat

9 Running the game

The GUI is written in C++, and there are builds available for both Linux and Windows. Although you could run the game without the graphics, it is not recommended, since it would be nearly impossible to tell what is going on.
These are the steps you take to run a game on the local machine.

1. *Start the GUI.* The executable for the GUI is named ps6gui.exe for Windows, and ps6gui for Linux. It can be run either from the command line or by double-clicking the ps6gui.exe file. Once the GUI is started, a blank field with the words "waiting for connection" will appear on your screen. By default, it will listen on port 3126 for a connection. It will remain blank until a SML program has connected to it.

2. *Start your SML program.* After you start the GUI, run your SML program (that includes the evaluator, the world, and the networking code that talks to the GUI). Go to your project directory and run `CM.make(‘‘sources.cm’’)` to compile the program. Then run `‘‘GameLoop.start(⟨host-list⟩, ⟨black team-job⟩, ⟨white team-job⟩)’’`, where ⟨host-list⟩ is a list of hosts (represented as hostname and port pairs), and ⟨black team-job⟩ and ⟨white team-job⟩ are strings representing file names you want to use as black and white teams. Alternatively, you can run `T.test()`, which starts up a game on localhost:3126.

10 Implementing a barrier abstraction in CL

Your third task is to use CL to implement a standard synchronization primitive called a *barrier*, corresponding to this SML specification:

(* A barrier is synchronization primitive for a group of n threads. Any thread from the group that reaches the barrier must block until all n threads have reached the barrier. Then all threads in the group may proceed. *)
type barrier
(* makeBarrier(n) creates a barrier for n threads. *)
val makeBarrier: int -> barrier
(* waitB(b) causes the current thread to block until the required number of threads have all called waitB(b). *)
val waitB : barrier -> unit

Your barrier implementation should be submitted to CMS as barrier.ch. Your file must be a CL header file that works with `#include`; if you put #include barrier.ch at the top of another CL file and run it, it should work and have access to all the barrier functions.

11 Written Problem

In addition to the game and bot implementation tasks described above, this project also includes a written problem on amortized complexity. This written question should be submitted to CMS, in .txt, .pdf, or .doc format.

A sorted array (or vector) is an appealing data structure for storing ordered data, because it offers the same $O(\lg n)$ lookup time as a balanced binary tree but has a compact representation and a good asymptotic constant factor. Unfortunately it doesn't support fast insertion.

A programmer named Mort has an idea for a fast mutable ordered set abstraction. Instead of storing all the elements in the sorted array, he will maintain a separate short linked list of up to $f(n)$ elements, where $f(n)$ is some function yet to be determined.

```
type set = {sorted: element array ref, recent: element list ref}
```

When the data structure is searched, both the list recent and the array sorted (of length n) are traversed. When an element is added to the data structure, it is appended to the list in constant time. If the recent list becomes longer than or equal to $f(n)$ elements, the $f(n)$ elements are sorted using mergesort and then merged in linear time with the n elements, which are already in order.

a. What is the complexity of a single lookup on this data structure, expressed as a function of $f(n)$ and n? To achieve complexity $O(\lg n)$, as with a balanced binary tree, what should Mort set $f(n)$ to?

b. The goal with this structure was to make inserts cheaper. As a function of $f(n)$ and n, what is the complexity of $f(n)$ inserts into this structure, starting from an empty recent list? (This should cause exactly one sort and merge)

c. We can reduce both insert and lookup to an amortized complexity of $O(\sqrt{n})$. Your goal is to prove this bound using potential functions. Recall that the amortized complexity of an operation changing structure s to s' is defined as the actual cost of operation plus $\Phi(s') - \Phi(s)$.

   Provide a Φ and a definition of f, and use them to show that the complexity of one lookup is $O(\sqrt{n})$ and the amortized complexity of one insert is $O(\sqrt{n})$.