CS 312 Project 2: λ-Quidditch

Assigned: November 15, 2007

Final submission due: 11:00PM, December 4, 2007
Checkpoint meetings: November 26 and 27, 2007

---

1 Introduction

In the previous part of the project, you were asked to develop an interpreter for a concurrent programming language. This part will allow you to put that language to good use: you will develop a game called λ-Quidditch. In terms of programming, you will have to implement the world for this game (in SML), as well as the code for the players (in CL). We have provided some graphical support that you can use to display the progress of the game. You will keep the same partner you had for part I; consult Prof. Rugina if this is exceptionally problematic.

This project places few constraints on how you implement it. 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 have to start by laying out a design of how you want to build your system.

For this part there will be a *checkpoint meeting* halfway 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.

1.1 Source code

Source code for this project is available on the course web site.

1.2 Use of CL

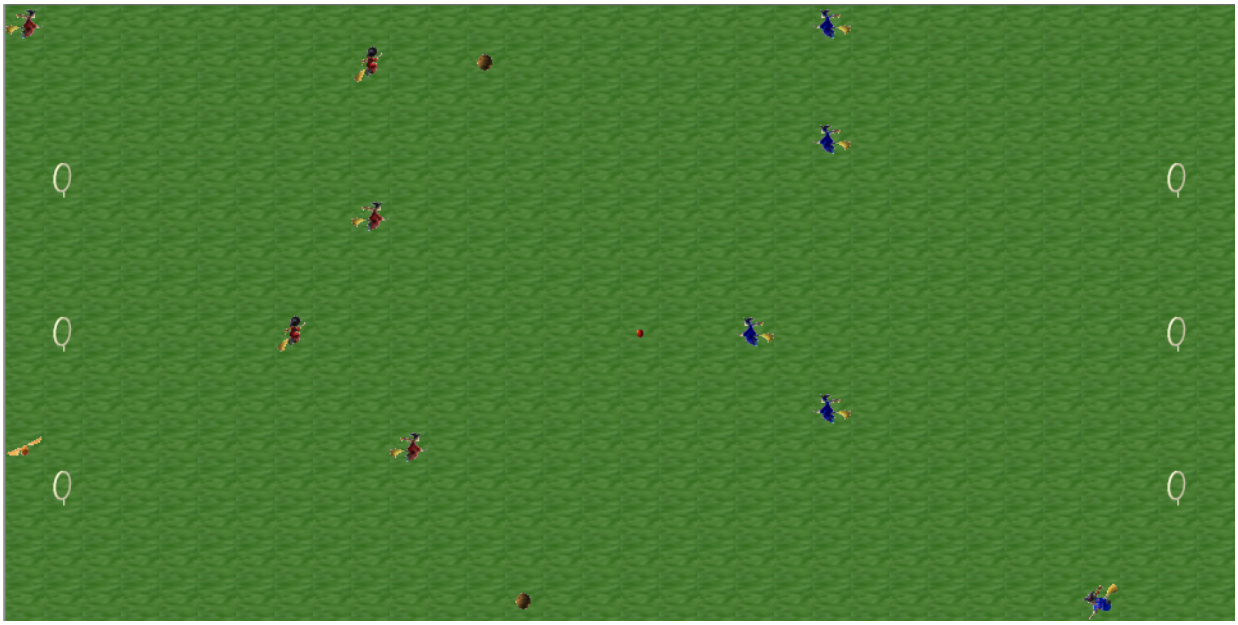
The game consists of two teams of players (or units). Each player will be driven by a program written in the CL language, defined in the first part of the project. You will implement not only the game (in SML) but also a team to play the game (in CL). Your evaluator from part I will run the CL programs of the players. You will need to copy your PS5 (Part I) code into the PS6 (Part II) distribution in order to compile it. You should not have to change your evaluator code except perhaps to fix bugs.

2 Game Rules

λ-Quidditch is similar to the fictional sport from the Harry Potter series. The game is played by two teams, the red team and a blue team, on a rectangular board with has two goals and several kinds of balls. The object of the game is to shoot one the balls in the opposing team's goal, capture another of the balls, and avoid being hit by yet another kind of ball. The teams that gathers the most points within a fixed amount of time wins the game.

Throughout this writeup, we will define things in terms of various constants, which will be typeset like this: AT_MOVE. These constants are defined in cl/constants.ch and in world/definitions.sml, and represent fundamental game values, such as the number of cycles it takes for a move (AT_MOVE), the maximum number of players on a team (MAX_PLAYERS), and so on.

2.1 The Field



- Size: The field is 32 squares long, 16 squares high.
- Cells: Cells on the field are identified by 2-dimensional coordinates, with (0,0) being in the upper left corner and (31,15) in the lower right corner. The first number identifies the left-right (x) coordinate, and the second number identifies the up-down (y) coordinate.
- Goals: Three one-square goal areas at each end of the board. The squares are not right next to the wall, and the ball moving into the square from any direction scores a goal. Goals are placed at positions (1,4), (1,8), and (1,12) for one team, and symmetric positions for the other team.

2.2 Directions

Each square has 8 neighbors. The units and arrows can move to any of a square's neighbors. However, a diagonal move will take longer since it covers more distance. The directions are numbered 0 to 7 as follows:

3	2	1
4	red	0
5	6	7

7	6	5
0	blue	4
1	2	3

2.3 Teams

There are two teams on the board, red and blue. The red team tries to protect the left goal areas and score in the right goal areas, and the blue team does the opposite. Although there are two teams, all robots are programmed in the same way, as if they were all in the red team, and were attacking to the right. It is the duty of the world to translate coordinates and directions for the blue team: a coordinate (x, y) becomes (BOARD_WIDTH-x, BOARD_HEIGHT-y) for the blue team, and a direction i becomes direction (i+4) mod 8 for the blue team. When processing actions that were requested by blue players or returning position or direction information to blue players, the world must perform these translations.

Each team consists of at most MAX_PLAYERS units.

2.4 Scheduling

The amount of time each process is allowed to evaluate is important to the fairness of the game. Moreover, the balls have to be given time to move when required. You are going to use your single-step evaluator from Project I to evaluate the CL code for the units. The order of the evaluation must be fair to both teams.

In every clock cycle, every unit of each team is stepped exactly once; the world is then notified that a cycle has ended to that it may update information and move the balls.

2.5 Stamina

Each player has a certain amount of stamina. A player's stamina is decreased by STAM_LOSS_ACTION for every move or kick that player performs. When a player's stamina falls below a certain threshold (defined by the constant STAM_THRESHOLD), their move and kick actions slow down (i.e., a greater delay is applied to them).

The penalty is applied to each move or kick action of that player by treating the base delay of those actions as (STAM_THRESHOLD - stam) * STAM_ACTION_PENALTY greater. For example, given STAM_THRESHOLD = 100, STAM_ACTION_PENALTY = 100, a unit with 93 stamina would have the base delay of its move and kick actions increased by (100-93)*100 = 700.

2.6 Spawning new players

There is a strict maximum to the number of players on a team (MAX_PLAYERS). Whenever a team has less than this number of units on the board, they may spawn new processes to create new units. Otherwise (if they are already at the maximum), spawn requests are denied. Newly spawned red units appear at a random position with x-coordinate 0. Newly spawned blue units appear at a random position with x-coordinate 39.

The stamina of a newly spawned unit is equal to STAM_STARTING - (nunits * STAM_SPAWN_DECREASE), where nunits is the number of units previously spawned by the team. If this would be negative, it is instead equal to 0.

2.7 Balls

- **Quaffle** (1 ball): The Quaffle is a normal ball. It can be picked up by moving onto it, carried around by players, and kicked by players. When kicked, it initially moves 1 square every AT_QUAFFLE cycles, but takes an additional AT_QUAFFLE_FRICTION cycles for every move past the first. If it goes into the goal of a team, the other team scores a point and the Quaffle reappears in the middle of the board.

  The Quaffle begins the game at QUAFFLE_INIT_LOC, at rest.

- **Snitch** (1 ball): The Snitch is an evasive ball. It moves at a high speed (faster than the players) and is unaffected by friction. The snitch makes one move every AT_SNITCH cycles. It moves by choosing a random direction and moving in that direction for SNITCH_MOVES_PER_SWERVE moves, or until it would run into any object (such as a player, goal, or other ball). When it has moved the indicated number of moves or encountered an object, it chooses a new random direction and repeats this process.

  The Snitch begins the game at a random position with x-coordinate between S_LOW_COL and S_HIGH_COL, with initial movement in a random direction.

  After the Snitch is captured, the game waits for AT_SNITCH_RESPAWN cycles. After that time, the Snitch respawns at a random position with x-coordinate between S_LOW_COL and S_HIGH_COL, with initial movement in a random direction.

- **Bludgers** (2 balls): The Bludgers are harmful balls. They move at a medium speed and are unaffected by friction. They make one move every AT_BLUDGER cycles. They move by going in their current direction for BLUDGER_MOVES_PER_SWERVE moves, then turning either left or right, at random, and repeating this process. When a Bludger runs into a player or a player runs into a Bludger, the player loses STAM_LOSS_BLUDGER stamina, and the Bludger moves to a new, random location.

The Bludgers begin the game at random positions with x-coordinate between B_LOW_COL and B_HIGH_COL, with initial movement in a random direction.

2.8 Collisions

A variety of different collisions may occur due to the movement of the balls. The results of such collisions are defined here. For results of collisions that result from movement of the players or other player actions, see the section on actions.

- If the Quaffle would move onto the square of a unit, that unit acquires the Quaffle.
- If the Quaffle would move onto the square of another ball or would move off the edge of the board, it bounces.
- If the Quaffle would move onto the square of a goal for a given team, the other team is given a point, and the Quaffle reappears at QUAFFLE_INIT_POS. If the appropriate square is occupied, the Quaffle instead reappears at one of the closest unoccupied squares in the same column.
- If the Bludger would move onto the square of a unit, that unit has its stamina decremented by STAM_LOSS_BLUDGER stamina, and the Bludger respawns in a random open location on the board, with its base velocity in a new random direction.
- If the Bludger would move onto the square of another ball, a goal, or would move off the edge of the board, it bounces.
- If the Snitch would move onto the square of a unit, another ball, or a goal, or would move off the edge of the board, it instead chooses a new random open direction to go in, resets its count of how many moves it should make in that direction to SNITCH_MOVES_PER_SWERVE, and makes its move in that direction.
- If the Snitch is ever in a situation where it is completely surrounded, it does not move.

2.9 Bouncing

When the Quaffle and the Bludgers hit certain objects, they bounce. This section details how bouncing works. Note that in all cases, no matter what the result of a bounce is, it still counts as a move for purposes of the Quaffle slowing down. For the purposes of this section, a "blocked" square is one that has an object in it that the ball would bounce off of. Be careful to note that the Quaffle and Bludger do not bounce off of all the same things. A goal square is blocked for a Bludger, but not for the Quaffle.

- When a ball is moving horizontally or vertically and bounces off an object, the position of the ball is unchanged, but the direction of its movement is reversed.
- When a ball is moving diagonally and bounces off the edge of the board, the balls movement is reflected off the wall at a 90 degree angle. For example, if the ball was moving northeast, got to (5,0), and then ran into the edge of the board, the ball would now be at (6,0) and moving southeast.
- When a ball is moving diagonally and bounces off the corner of something, if both of the squares next to the object are empty or both squares are blocked, the ball simply reverses direction. If one square is blocked but the other is not, this case is treated like bouncing off the edge of the board.

  For example, suppose the ball is moving northeast and encounters an object it should bounce off. If both the north and east positions are empty or if they are both blocked (forming a corner), the ball reverses direction. If the east position is blocked but the north position is not, the ball bounces at a 90 degree angle and is now moving northwest. If the north position is blocked but the east position is not, the ball bounces at a 90 degree angle and is now moving southeast.

2.10 Start and End

Start: Each team starts with one unit, on their end of the board (they can immediately spawn units up to the limit). Both initial units have STAM_STARTING stamina. The red team's initial unit is placed at (2,10), facing east. The blue team's initial unit is placed at (29,10), facing west. The Quaffle starts in the center of the board, at QUAFFLE_INIT_POS. The Bludgers start at a random position between the columns B_LOW_COL and B_HIGH_COL, and the Snitch starts at a random position between the columns S_LOW_COL and S_HIGH_COL.

Scoring: QUAFFLE_POINTS points are scored for a Quaffle goal. SNITCH_POINTS points are scored for capturing the Snitch.

End: The game ends after GAME_LENGTH clock cycles. At this time, the team with the most points wins. If both teams have the same number of points, the game is a draw.

2.11 Actions

The units interact with the world by using the `do` command. Each action takes a certain number of clock cycles. A *clock cycle* is a single evaluation step for all of the units. In other words, if a unit performs an action that takes 100 clock cycles, it takes the time of that specific unit performing 100 evaluation steps, not $\frac{100}{\text{No. of units}}$ steps.

Here is a description of the possible actions.

- **move**: The unit moves from one square to an adjacent square in the direction it is currently facing. Units can move only one square at a time.
  - If a unit tries to move onto another unit and neither unit has the ball, the move fails. If one unit does have the ball, let x be the x-coordinate of the unit with the ball, and let g be the end of the board for that unit's team (0 for red, 31 for blue). With probability $|x - g|/\text{BOARD\_WIDTH}$, the unit without the ball successfully gains control of the ball. If the moving unit loses the fight, both units maintain their position. If the moving unit wins the fight and there is an open square behind the other unit in the direction that moving unit moved, the other unit is pushed back into that square, and the moving unit moves into the square the other unit used to be in.
  - If a unit moves onto the Quaffle when no other unit has it, the move is successful and the unit now has the Quaffle.
  - If a unit tries to move onto a Bludger, the move fails and the unit stays where it is. Additionally, the unit loses stamina, and the Bludger respawns in a random location, just as if the Bludger had hit the unit due to the Bludger's movement.
  - If a unit moves onto the Snitch, the move is successful and the Snitch is captured. The unit's team gets SNITCH_POINTS points and the Snitch is removed from the board until it is time for it to respawn.
  - If a unit tries to move off the edge of the board or into a goal area, the move fails.
- **turn**: The unit turns from his current direction to either the left or the right. A turn never fails; if the direction passed in is invalid, the unit turns left.
- **kick**: The unit attempts to kick the Quaffle in the direction it is facing. This action fails if the unit does not have the Quaffle.
  - If the square in front of the unit is off the edge of the board or another ball, the kick fails.
  - If the square in front of the unit is another unit, a one-square pass is performed and the other unit immediately gains control of the Quaffle.
  - If the square in front of the unit is a goal square, the Quaffle immediately goes in, with all the attendant consequences detailed in the secion on ball collisions.

  - Otherwise, in the case where the square in front of the unit is empty, the unit loses possession of the ball, and the ball is now in that square, with initial velocity equal to AT_QUAFFLE. The Quaffle then moves in a straight line as detailed in the section on the balls.

- **my status**: Returns the current location and direction of the unit.
- **team status**: Returns the current number of units and score of each team.
- **ball status**: Returns the position of the balls.
- **scan**: Returns the objects next to the unit in each of the eight directions.
- **look**: Returns the first object and the distance to that object in some direction from the position of the unit.
- **inspect**: Returns the object at a particular location.
- **get time**: Returns the number of clock cycles until the end of the game.
- **talk**: Displays text to the chat area in the GUI.

3 Implementing actions

The units have a list of actions that they can take via the `do e` command. Each action takes a specified amount of time to execute. Handling the time actions take is done through `delay` e `by` n expressions, as described in Part I of the project. A unit that calls an action gets returned a value wrapped in a `delay` expression, which produces the required delay.

Each action returns an *Action-Specific Return* (ASR), as described in subsequent sections. The results of an action should be visible to the world and the other units when the action is performed and before the unit is made to wait. Note that in the table that describes all ASR's, the square bracket notation $[x, y, z]$ denotes lists, whereas the parenthesis notation (a, b) denotes pairs. See Section **??** for a description of how lists are represented in CL.

Actions that don't return something specific should return R_SUCCESS if nothing went wrong, R_FAIL if the action failed. A bot moving into another bot with the ball and successfully taking it results in R_WON.

3.1 Possible actions

The figures on the following pages describe the possible actions in more detail. Here are some of the constants mentioned in the actions:

T_EMPTY	Empty spot
T_WALL	A wall
T_REDGOAL	Own goal
T_BLUEGOAL	Opposite goal
T_PLAYER	A team player
T_OPLAYER	An enemy player
T_PLAYERWBALL	A team player with the Quaffle
T_OPLAYERWBALL	An enemy player with the Quaffle
T_QUAFFLE	The Quaffle
T_SNITCH	The Snitch
T_BLUDGER	A Bludger

These and other constants are defined in cl/constants.ch and world/definitions.sml.

Command	Base Time	Args	Description	ASR	
A_MOVE	AT_MOVE	None	Move forward	R_SUCCESS	if the move was successful
				R_WON	if a unit tried to move onto an enemy player with the ball and took the ball
				R_FAIL	otherwise
A_TURN	AT_TURN	Int i	Turn in direction i, where i = T_LEFT or T_RIGHT.	R_SUCCESS	in all cases. If the direction passed in was invalid the unit will turn left.
A_KICK	AT_KICK	None	Kick the Quaffle forward	R_SUCCESS	if the kick succeeded
				R_FAIL	if the kick failed
A_MYSTAT	AT_MYSTAT	None	Returns the status of the unit	$[(x, y), d, s, q]$	where (x, y) is the unit's location, d is the direction the unit is currently facing, s is the unit's stamina, and q is 1 if the unit has the Quaffle, 0 otherwise
A_TEAMSTAT	AT_TEAMSTAT	None	Returns the status of the unit's team and the number of units on the enemy team	$list$	where $list$ is $[n, no, s, so]$, n and no are the number of units on the player's and the enemy's team respectively, and s and so are the score of the player's team and the enemy's team respectively
A_SCAN	AT_SCAN	None	Returns the items in the eight cells around the unit.	l	Where l is of the form [n,ne,e,se,s,sw,w,nw]. n is the object in the north cell, ne is the object in the northeast cell, etc.
A_LOOK	AT_LOOK	Int i	Returns the nearest object in direction i, where i = DIR_N, DIR_NE, DIR_E, etc.	[o, d]	where o gives the type of the first object, and d is the distance to that object.
A_INSPECT	AT_INSPECT	(Int x, Int y)	Returns the type of the object at location (x, y)	o	where o gives the type of the object at the specified location.
A_BALLSTAT	AT_BALLSTAT	None	Returns the status of the balls	$list$	where $list$ is $[q, s, b_1, b_2]$, q is the position of the Quaffle, s is the position of the Snitch, and b_1 and b_2 are the positions of the Bludgers (if the Snitch is currently not on the board, s is $(-1, -1)$)
A_TALK	AT_TALK	String s	Prints s in the text window	R_SUCCESS	

Figure 1: List of Actions

4 CL Extensions

We point out that several CL features and patterns that you might find useful when programming your units. Note that *none* of these features require you to change your evaluator.

4.1 Recursive functions

We have added support in the CL language for recursive functions. You can define recursive functions by using the keyword "rec", as in the example below:

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

The parser automatically expands recursive function definitions into an equivalent piece of code that implements recursion using references. For the above program, the parser automatically generates an AST that corresponds to the following program:

```
let __fact = lref 0
let fact = __fact := (fn n => = if n=0 then 1
                                else n * ((!__fact) (n-1))
in fact 3
```

The above translation requires no new AST nodes, so the resulting code requires no changes in the evaluator. Note, however, that a new variable `__fact` is being introduced. Make sure that your code does not use variable names that begin with double underscores, as they might conflict with the variables automatically generated by the parser.

4.2 Includes

You may wish to write code that multiple CL programs can use. You can now 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 List library

We have provided a useful library for manipulating lists, located in the file cl/lists.ch. The functions in the list library are similar to those in SML, as shown below. Lists are represented as either 0 (for the empty list) or as a pair containing an item and the representation of another list. For example, the list $[1, 2]$ is represented as "(1,(2,0))". In general, a list with elements $x_1, x_2, \ldots, x_n$ is represented in CL as $(x_1, (x_2, (\ldots (x_n, 0) \ldots)))$. The list library defines the following

functions:

`cons x y`	Add the item `x` to the head of the list `y`
`nil`	Return an empty list
`length l`	Return the length of the list `l`
`nth l n`	Return the n-th list `l`
`foldl f acc l`	Fold the function `f` over the list `l` with the initial accumulator `acc`, starting with the first item
`foldr f acc l`	Fold the function `f`, which is a curried function taking an item and the accumulator, over the list `l` with the initial accumulator `acc`, starting with the last item
`map f l`	Map the function `f`, which is a curried function taking an item and the accumulator, over every item in the list `l`
`append l1 l2`	Append `l1` to the front of list `l2`

All of the above functions are implemented using the `rec` construct.

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

5 Java GUI

Your world program will talk to a Java program that graphically represents the game. Once the connection to the GUI is established [1], your program will interact with the GUI using one command:

```
NetGraphics.report( msg )
```

where "msg" is a message that describes an event on the field. This is a string consisting of an event name, followed by a number of parameters, separated by spaces. The following events are recognized by the GUI:

- `"[CONNECTING]"`, initializes the connection with the GUI.
- `"set <image> <x> <y>"`, where `<image>` is a string describing the object image (below is a list of possible images), and `<x>`, `<y>` are two coordinates where the object must be added.
- `"set <image> <x> <y>"` where `<image>` is a string identifying an image, and `<x>`, `<y>` are the images coordinates.
- `"score <red score> <blue score>"` updates the score.
- `"time <t>"` updates the time left in the game to `<t>`.
- `"name <color> <name>"` updates the `<color>` team's name display to be `<name>`.
- `"chat <color> <message>"` prints message `<message>` from team `<color>`.

[1] This is done by invoking "NetGraphics.setup", with a list of pairs (*host*, *port*) as arguments. For instance, `NetGraphics.setup [("localhost", 3120)]` connects to the port 3120 on your machine. Function `NetGraphics.setup` is called from `/world/loop.sml`, which starts the main loop of the game. This file is provided to you.

We require <x> and <y> to be valid coordinates and <img> to be a valid image name. The list of possible image names for <img> are listed below, where color is either "R" or "B" if the unit is on the red team or the blue team, respectively.

Image	Description
player[q]CD	Draws a player of color C facing direction D. Color C is one of B or R. Direction D is an integer in [0,7]. Letter 'q' is optional, indicating whether the player posseses the quaffle.
quaffle	Draws the quaffle.
snitch	Draws the snitch.
bludger	Draws the bludger.
post	Draws the goal post.

A few sample commands that can be passed to "NetGraphics.report()" are shown below:

```
"set empty 3 7"
"set playerqB1 4 7"
"set snitch 3 3"
```

When a unit moves, you have to restore the information in the old position. Then, set the new position for the unit. It is important that you restore the old positions all at once, and then set the new positions. Otherwise, restoring the old position of a unit may erase the image of another unit that just moved.

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. 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.

6.2 Designing the world

Your first task is to implement the λ-Quidditch world in the files `world/action.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 units take are rendered in the graphic display using the interface detailed in Section 5. You can use the sample unit program we provide to test your world.

6.3 Designing an team

Design a CL team in a file `cl/team.cl`. Your team should be able to consistently beat the team provided by the course staff. You will be graded on the number of times your team beats ours and the strategy which you use.

6.4 Documentation

As with all of the assignments up to this point, you should submit some documentation regarding your project. Since this project is quite open-ended regarding the way one may choose to implement it, documentation becomes even more important. In your documentation, you should discuss all of the following:

- **Implementation decisions**: Justify the modules into which you broke down your code, including specific data structures you chose to use. Much of this information may come from your design document submitted at checkpoint time. If your strategy changed between the design document and your implementation, explain why.
- **Specification changes**: If refinements of the specifications given in the project are necessary, described these changes and justify them. With such a complex program to implement, there are some things that may be somewhat ambiguous. Any such ambiguities brought to the attention of the course staff are clarified in this writeup and often in the newsgroup. You will be responsible for making sure your program conforms to these clarifications; resolving these problems in a different way will result in a loss of points. However, any ambiguities we do not clarify, please implement them as you see fit and document them.
- **Validation strategy**: Report how you validated your implementation. Explain and justify your testing strategy.

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 what is going on in the world and what is going on in the graphics match.** 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 may actually be problems with the units.** If you are using your own units to test the actions and something seems wrong, the units could just as easily be at fault.
- **It is best to 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 the easier actions and work up to the harder ones. An actions like `turn` is probably easier to implement relative to the other actions.

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

6.6 Checkpoint meeting

For this assignment, there will be a *checkpoint meeting* halfway through the assignment. These meetings will be held on November 26 and 27. You are expected to 1) explain the design of your system and give a brief description of the design of your CL units and 2) hand in a printed copy of a signatures for each of the modules in your design.

Moreover, we strongly encourage that you to come discuss your design with the course staff during consulting/office hours before and after the meetings.

6.7 Final submission

You will submit: 1) a zip file `project.zip` of all files in your `project` directory, including those you did not edit; and 2) your documentation file `doc.txt` (or `doc.pdf`). 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`.

Your submission should unzip a `/project` folder, which contains your `sources.cm` and all of the other directories. We expect to be able to unzip your submission, and run `CM.make()` in the newly created `project` directory to compile your code without errors or warnings. *Note: Submissions that do not meet this criterion will be docked points.*

7 Tournament

Sometime during finals period, most likely on December 7, 2007, we will hold a competition between the CL players of the students who wish to compete. Each group may submit a CL team that will play against other students teams. Details on the tournament time and location and the submission procedure will be available later.

8 Given Files

Many files are provided for this assignment. Most of them, you will not need to edit at all. In fact, 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 functions.

`world/action.sig`	Signature file for handling actions and interacting with the world. Note that this file has been changed from PS5
`world/action.sml`	Functions for handling actions and interacting with the world
`world/game.sml`	Handles the game state
`world/game.sig`	Signature file for the functions contained in the game
`world/loop.sml`	Starts and continues the main game loop
`net/*.sml`	Network SML code for communicating with the GUI
`gui/*.class`	The GUI class files
`gui/gfx/*`	Graphics files for the GUI
`cl/*.ch`	Libraries for lists and booleans
`cl/sampleunit.cl`	Sample team program

9 Running the game

The Java GUI class files are compiled with the latest version of Java. Fell free to recompile them if you use a different Java version. 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'd take to run a game on the local machine.

1. *Start the GUI.* Start a command prompt (in Windows) or a terminal (in *nix). To start a command prompt in Windows, click on the Start menu, go to Run and type in `command`. At the command prompt, go the the `project/gui` directory and run "`java -jar Gui.jar <host> <port>`". This tells the graphics program to start up and wait for a connection to the SML world, on `<host>` at port `<port>`. The `<host>` and `<port>` parts are optional. Runing "`java -jar Gui.jar`" is equivalent to running "`java -jar Gui.jar localhost 3120`". Once the GUI is started, a field will pop up on your screen. The field will remain empty until the SML program has connected to it.

2. *Start your SML program.* After you started the GUI, run your SML program (that includes the evaluator, the world, and the networking code that talks to the GUI). Go to the `project` directory and run `CM.make()` to compile the program. Then run "`Game.start(<host-list>, <red team-job>, <blue team-job>)`", where `<host-list>` is a list of hosts (represented as hostname and port pairs), and `<red team-job>` and `<blue team-job>` are strings representing file names you want to use as red and blue teams. Alternatively, you can run `T.test()`, which starts up a game on `localhost:3120`.

The game should now begin. If at any point you need to recompile and start the game over, you will also need to restart the graphics program.