CS 312 Problem Set 6: Multi-Ball

Assigned: April 16, 2005

Final submission due: 11PM, May 6, 2005
Checkpoint submission: 11PM, April 27, 2005

---

1 Introduction

In problem set 5, you were asked to develop an interpreter for a concurrent programming language. This problem set will allow you to put that language to good use: you will develop a robotic soccer game called Multi-Ball. In terms of programming, you will have to implement the world for this game (in SML), as well as the code for your players (in CL). We have provided some graphical support that you can use to display the progress of the game graphically. You will keep the same partner you had for PS5; consult the course staff if this is exceptionally problematic.

This problem set places few constraints on how you implement it. This does not mean you can abandon what you've 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 problem set there will be a *checkpoint submission* halfway through the assignment. You are required to submid a document that describes the design of your system and as much of the remaining code as you have by that point.

1.1 Source code

Source code for this project is available in CMS.

1.2 Clarifications and changes

- [Apr 20]: Typo: the GUI command for showing credits (Section 5) is "`credits`", not "`credit`".
- [Apr 17]: The specification for directions (Section 2.3 and Section 2.4) has been changed. There is one single direction for all players, both red and blue.

1.3 Use of CL

The game will be played by bots driven by programs written in the CL language. You will implement not only the game but also at least one bot program that plays the game. Your PS5 evaluator will run this program. You will need to copy your PS5 code into the PS6 distribution in order to compile it. You should not have to change your evaluator code except perhaps to fix bugs.

2 Game Rules:

Multi-Ball is a game played by two teams of CL robots, the red team and the blue team. The object of the game is to shoot the ball into the opposing team's goal, scoring a point, as many times as possible. The team which has accrued the most points in this fashion shall win the game. On the field are two balls which may each be used for this purpose.

2.1 The Board

Multi-Ball is played on a rectangular field 20 cells wide by 40 cells long. Each cell is a square in which a robot, ball, etc may exist. The goals shall be columns 12 cells tall, placed immediately outside the playing field on opposite ends, centered vertically. Robots may not occupy any part of the goal. Cells of the board are identified by 2-dimensional coordinates, with (0,0) being in the upper left, and (39, 19) being in the lower-right. The first number identifies the left-right (x) coordinate, and the second number identifies the up-down (y) coordinate.

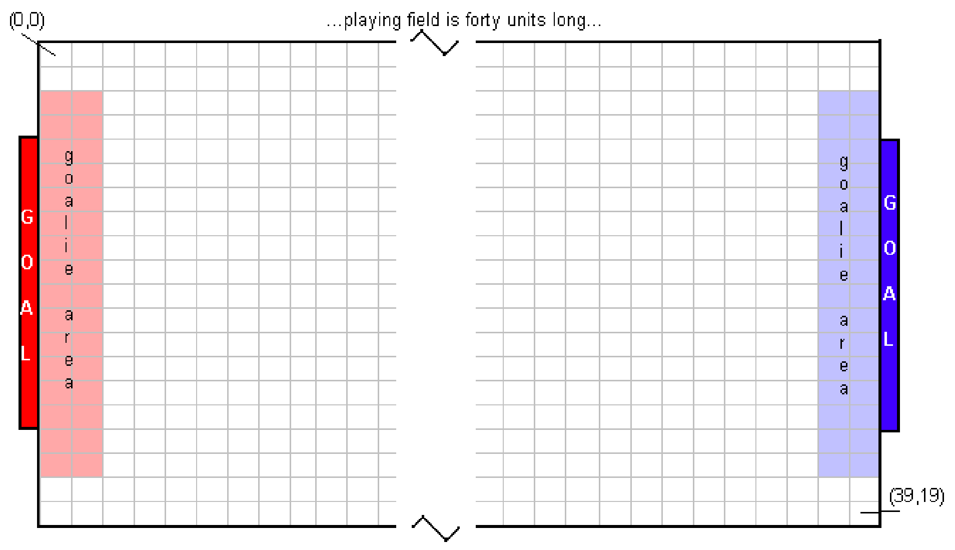


Figure 1: The Multi-Ball board

2.2 Goalie Area

The cells closest to the goal are reserved – only that team's goalie may move within these cells. The red (left) team's goalie is the only robot allowed in the cells in a rectangle extending from $(0, 2)$ to $(1, 17)$. The blue team's goalie is the only robot allowed in the cells in a rectangle extending from $(38, 2)$ to $(39, 17)$.

2.3 Directions

Each cell can touch eight neighboring cells. Robots and balls may move in any of the eight directions, though moving diagonally takes longer than moving in one of the four cardinal directions (as it covers more distance). The eight possible directions are numbered from 0 to 8. Direction zero means facing right:

3	2	1
4	●	0
5	6	7

2.4 Objects in the game

There are two teams on the board, red and blue. The red team tries to protect the left goal and to score in the right goal, while 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 $(39 - x, 19 - y)$ 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 robots, the world must perform these translations.

The board also has two balls, which may be carried by bots or may be in motion. The balls have a location and a direction in which it moves if it is not stationary or possessed by a bot, as well as a bit to indicate the team which last shot it.

Each team has one goalie and possibly one or more players. The goalies and players all have locations and directions of their own. The players can move around the board, intercept the ball, fight for position, and try to score goals. Each of these actions takes a certain number of clock cycles. A *clock cycle* is a single evaluate step for all of the bots on both teams. In other words, if a bot performs an action that takes 100 clock cycles, it takes the time of that specific bot performing 100 evaluation steps, not $\frac{100}{\text{No. of bots}}$ steps.

2.5 Starting the game

When the game starts, the balls are stationary at board coordinates $(19, 10)$ and $(20, 10)$. Each team is controlled by an CL program. The first CL process that starts for each team is registered as the goalie. The goalies start at positions $(0, 10)$ and $(39, 10)$ for red and blue respectively. The goalies cannot leave their designated "goalie areas." If the goalie process is killed for any reason, no other bot is allowed to take its place and the team must perform without a goalie.

2.6 Spawning new bots

After the game start, both teams can spawn more bots using the `spawn` command in CL. The goalie is the only bot that can directly spawn more bots. A bot spawned appears in a randomly-chosen free tile in front of the goalie area. When a bot is spawned, the number of spawn credits reduces by a number given in a table below. If a team does not have enough spawn credits, then it cannot spawn a new teammate. Initially, each team is assigned six spawn cretdits, so that the goalie may spawn two bots. It is important to note that a bot does not necessarily know if the team has enough spawn credits and therefore might try to spawn a bot despite the fact that the team has already spawned all its additional players. In this case, the world should catch this fact and immediately terminate the new process. The bot that attempted to spawn proceeds normally.

Robots on the team:	Credits required for next robot:
1-3	3
4	6
5	9
6	12
7	15
8+	18 + 3 * (number of Bots - 8)

2.7 Ball movement

The balls start at coordinates (19, 10) and (20, 10) and stay stationary until a bot picks a ball up. Once picked up, a ball moves with the bot possessing it. The ball can move on its own once it is kicked by a bot (as described below). When the ball is not possessed by a robot, it moves in the direction it was kicked until one of the following things happens:

1. **It comes in contact with a bot not carrying a ball.** If the ball enters a tile where a bot currently sits, that bot captures the ball. The ball moves with that bot until the bot kicks it or another bot steals the ball. If a goalie controls the ball for more than 250K clock cycles, the ball leaves its possession and returns to the center of the board. (19,10) for the red goalie, and (20, 10) for the blue goalie. If the appropriate tile is occupied already, the closest neighboring tile is used instead.

2. **It goes into the goal.** A ball goes into the goal when it enters a goal tile. The coordinates of the tiles making up the goal area on the red (left) side are $(-1, 4)$ through $(-1, 15)$. On the blue side, the goal is cells $(40, 4)$ through $(40, 15)$. When the ball goes in the goal, the team opposite of the goal into which the ball went earns a point. It is possible for a robot to kick the ball into its own goal, scoring a point for the other team. After the goal is scored, the ball resets to the center of the board (coordinates $(19, 10)$ or $(20, 10)$ depending on whether it was scored in the red or blue goal, respectively, and stays stationary as at the start of the game. All of the bots remain where they were when the goal was scored and continue executing their programs. If one of those two cells is occupied, the world places it as close to those two cells as it can by finding a free neighboring cell.

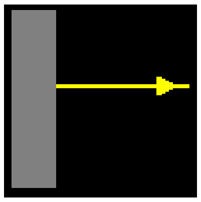
3. **It runs into the wall.** A ball runs into the wall when it tries to enter a tile beyond the top or bottom of the board, e.g., y-coordinates -1 or 20. When the ball runs into a wall with a diagonal direction, it bounces out with the expected "opposite" diagonal direction. For instance, if the ball is moving up-right when it hits, it will bounce in the down-right direction. If the ball is kicked straight up or straight down into a wall, it ricochets off in a

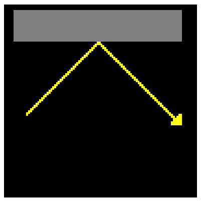
diagonal direction toward the goal of the team opposing the kicker. Thus, if a blue robot kicks the ball straight up into a wall, it bounces down-left toward the red goal. If a ball hits a goal-wall head on, it bounces straight back off the wall.

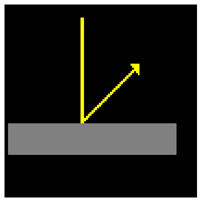
4. **It runs into a robot already holding a ball.** Robots may not capture more than one ball. In this case, the robot acts as a wall, and the same collision effects occur.

5. **It runs into a spawn credit.** The spawn credits are physical objects which the robots must pick up by running on top of them. Thus, the ball bounces off of spawn credits just as though they were walls.

6. **It collides with another ball.** The two balls bounce off of each other as though an elastic collision occured: they swap their motion and velocities. Thus if ball A moves right, and ball B moves straight up when they collide, ball A will move up, and ball B will move right after the collision. This also applies to the case when one of the balls is stationary: the stationary ball starts moving with the speed and direction of the old moving ball, and the moving ball becomes stationary.


Collision with a goal-wall


Angular collision with the top wall


Collision with the bottom wall, as kicked by the red team

The collision of a ball with the corner of a wall-like object (e.g., wall, spawn credit or robot holding a ball) works according to the following rules. Suppose that the ball is traveling north-east and hits a wall in its path. If both the north and east positions are empty, or if they're both not empty (a corner), then the ball reverses it's direction. If one is empty and the other is not empty, then this is a wall case and the ball's direction becomes the appropriate 90 degree reflection.

When the ball is moving on its own, it should slow down gradually. When the ball is set in motion, it moves once every 3000 clock cycles. After each move, this number increases by 500. Since robots take approximately 5000 clock cycles to move, the ball starts off moving faster than the bots and then after about six or seven moves slows down, allowing bots to catch up to it.

2.8 Spawn credits

Bots may be spawned by the goalie at any time, but only if the team has enough spawn credits. The spawn credit tallies are maintained by the world. A team gains spawn credits by having robots move into cells where spawn credits are. Spawn credits are randomly placed about the board by the world, but they cannot appear on a tile which is already occupied by any other item (ball, bot or spawn credit). They also cannot appear in the goalie area. When spawn credits are consumed by robots, the world reintroduces the credits after a delay. There are a maximum of twenty spawn credits on the board at any time. Individual robots on a team do not have their own spawn credit counts; there is one counter per team.

2.9 Bot actions

Bots interact with the world by using the `do` command. The fundamental actions the bots can take are **move**, **turn**, and **kick**. In addition, a bot can request information about the game state and ball position. Here is a description of these actions.

move	A bot moves from one tile to an adjacent tile in the direction it is currently facing. The bot can only move one tile at a time. Only bots registered as a team goalie can move within the designated “Goalie Area.” If a bot that is not a goalie attempts to move into the goalie area, or if any bot attempts to move into a wall, the move fails and the bot stays in its current location. Additionally, the goalie can only move within the Goalie Area. Only one bot may occupy a tile at any given time. If a bot tries to enter a tile containing either a teammate or an opponent, it fights that bot for the right to be there. The probability p that the moving bot will win the fight depends on the size of the teams. Let Bot A be on a team with x bots and let Bot B be on a team with y bots. If A tries to enter a tile with B already in it, then the chance that A will win the fight is $\frac{cy}{x+y}$, where c is $\frac{1}{2}$ if neither bot has a ball, $\frac{1}{2}$ if both bots have a ball, $\frac{1}{4}$ if only A has a ball, and 1 if only B has a ball. The outcome of the fight depends on these probabilities and on whether or not one of the bots has a ball. If exactly one of them had a ball, the winner of the fight gains possession of the ball. If both had a ball, ball possessions are not changed by the fight outcome. If B won the fight, then B stays in the tile and A doesn’t move from the tile it was originally on. In addition, A is stunned for a period of time equal to twice the normal move time. If A won the fight then A tries to push B in the direction A was moving. The push is allowed if B is allowed to move into the tile it was pushed to (i.e. the tile is not a wall or in the goalie area) and there are no existing bots, balls or spawn credits in that tile. If the push is allowed, then A moves into the tile it was aiming for and B is pushed into an adjacent tile. If the push is not allowed then both the bots remain in their original tiles; the only change in this situation is a ball possession change if applicable. Note that no fights can ever occur between goalies and other players, as it is illegal for them to move into each other’s areas.
turn	A bot turns from the current direction to either the left or to the right and faces the next side of the tile. A turn never fails; if the direction passed in is invalid, the bot turns left.
kick	A bot kicks the ball in the direction it is currently facing. When a ball is kicked, it leaves the bot’s tile and travels to the tile immediately in front of the bot in the direction the bot is currently facing. A kick fails if the bot does not have the ball, if the bot is currently facing a wall, or if the tile in front of the bot contains a bot that already has a ball, a ball, or a spawn credit. If the tile in front contains a bot who does not have the ball, that bot receives the ball - a one-tile pass occurs and the kick succeeds. If the kick succeeds, the bot continues executing as it would have. If the kick fails, the bot keeps the ball and continues to execute.
mystatus	returns the current status for a bot. This includes the coordinates of the bot, whether the bot has the ball or not, and the direction the bot is currently facing.
ballstatus	returns the coordinates of both balls.
look	returns the first non-empty object in a direction from where a bot is.
inspect	returns the object at a particular location.
nearest	returns the location of the nearest object of a particular type.
teamstatus	Returns the number of spawn credits on the bot’s team, the number of bots on the team, the number of bots on the opposing team, the bot’s team’s score, the opposing team’s score, and the list of the PIDs of all robots currently alive on the bot’s team.

2.10 Scheduling

The amount of time each team is allowed to evaluate is important to the fairness of the game. Moreover, the ball has to be given time to move on its own when required. You are going to use your single-step evaluator from Problem Set 5 to evaluate the CL code for the bots. The order of the evaluation must be fair to both teams.

In every clock cycle, every bot on each team is stepped exactly once; the world is then notified that a cycle has ended, so that it may update information, for instance, move the ball.

The winner of the game is the team with the most points, determined by a length of the game. For your purposes, do not worry about how long the game runs.

3 Implementing actions

The bots 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 Problem Set 5. A bot that calls an action gets returned to it the return 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 bots when the action is performed and before the bot is made to wait.

3.1 Possible actions

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

T_EMPTY	Empty spot
T_BALL	A ball (either one)
T_TEAMMATE	A teammate
T_OPPONENT	An opponent
T_TMGOALIE	Team's goalie
T_OPPGOALIE	The opponent's goalie
T_TEAMMATEWITHBALL	An opponent with a ball
T_OPPWITHBALL	An opponent with a ball
T_TEAMGOALIEWITHBALL	The teams's goalie with a ball
T_OPPGOALIEWITHBALL	An opponent's goalie with a ball
T_PSPAWN	Spawn credit
T_WALL	The edge of the board (a wall)
T_OPPGOALA	The opponent's goal area (T_BALL if a ball is there)
T_TMGOALA	The team's goal area

Figure 2: Return values in the adjacency list

These and other constants are defined in cl/constants.ch.

Command	**Base Time**	**Args**	**Description**	**ASR**	
A_MOVE	AT_MOVE	None	Move the bot forward	R_SUCCESS	if the move was successful
				R_HITWALL	if the bot tried to move into a place where it cannot
				R_WON	if the bot tried to move into a position with another bot who had the ball and won control of the ball (may or may not have moved)
				R_LOST	if the bot tried to move into a position with another bot and lost a battle. If this is the case, the action should twice as long as normal (a “stunned” period).
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 (neither T_LEFT nor T_RIGHT) the bot will turn left.
A_KICK	AT_KICK	None	Kick the ball forward	R_SUCCESS	if the kick was successful
				R_HITWALL	A wall or any other object was in front of the bot, except if the object was a bot not possessing the ball (see section 2.9).
				R_NOBALL	The bot did not have the ball
A_MYSTAT	AT_MYSTAT	None	Returns the status of the bot	$[(x, y), hb, d]$	where x, y are the bot’s coordinates, hb is a boolean specifiyng whether or not the bot has the ball ($1 = true,\ 0 = false$), and d is the direction the bot is currently facing.

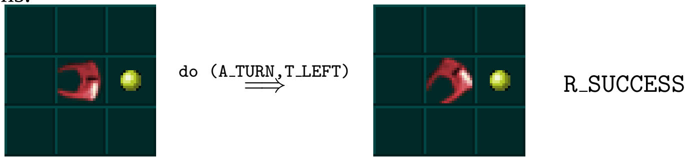
Figure 3: List of Actions

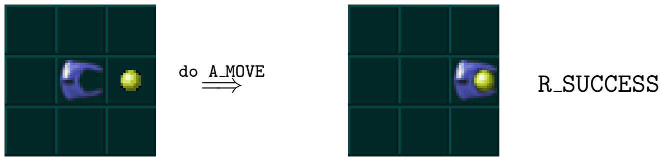
Command	Base Time	Args	Description	ASR	
A_BALLSTAT	AT_BALLSTAT	None	Returns information about the position of both balls	$((x_1, y_1), (x_2, y_2))$	where (x_1, y_1) are the coordinates of the first ball, and (x_2, y_2) are the coordinates of the second ball.
A_LOOK	AT_LOOK	Int i	Returns the nearest object in direction i	T_BALL, etc.	where the return value gives the type of the first non-empty object in this direction.
A_INSPECT	AT_INSPECT	(Int x, Int y)	Returns the type of the object at location (x, y)	T_BALL, etc.	where the return value gives the type of the object at the specified location.
A_NEAREST	AT_NEAREST	T_BALL etc	Find the closest item of the type specified, using Cartesian distance as the metric.	(x, y)	which are the x and y coordinates of the nearest item of this type.
				R_FAIL	if no object of the specified type was found on the board.
A_TEAMSTAT	AT_TEAMSTAT	None	Returns information about the status of both teams	l	where l is of the form: $[c, numb_1, numb_2, s_1, s_2, p]$ where c is the number of spawn credits on the bot's team, $numb_1$ is the number of bots on the bot's team, $numb_2$ is the number of bots on the opposing team, s_1 is the score of the bot's team, s_2 is the score of the opposing team, and $p = [p_1, \ldots p_k]$ is a list of the PIDs of all alive bots on the bot's own team.
A_TALK	AT_TALK	String s	Prints s in the text window	R_OK	

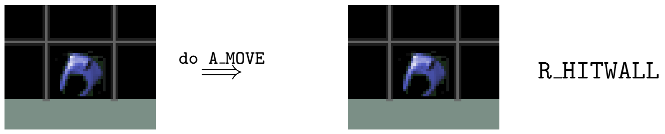
Figure 4: List of Actions, continued

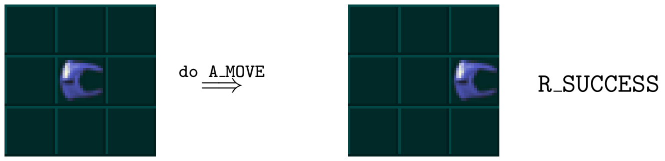
3.2 Examples of actions

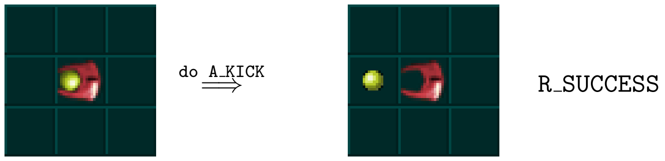
What follows are some graphical representations of what would happen and the return values given a bot calling certain actions.

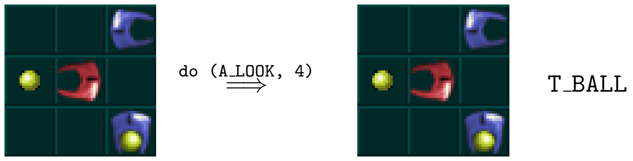












4 CL Extensions

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

4.1 Recursive functions

There is no explicit language support for recursive functions. However, these can be easily written using references to functions, as discussed in class. For instance, consider the factorial function:

```
let fact x = if x=0 then 1
             else x * (fact (x-1))
in fact 3
```

This function can be implemented in CL as follows:

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

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 ps6 directory and want to include in a bot the constants.ch file 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 botfunctions.ch with `#include botfunctions.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`
`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 accumlator, 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 accumlator, over every item in the list `l`
`append l1 l2`	Append `l1` to the front of list `l2`

Many of these functions are recursive. Their implementation takes advantage of the pattern-matching construct `typecase`. For instance, the implementation of function `length` is as follows:

```
let length' = lref 0 in
let length = length' := (fn l =>
  typecase l of
    (h, t) => 1 + ((!length') t)
  | any x => 0)
```

4.4 Boolean library

We also provide a few values and functions for representing and manipulating booleans:

`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 board. 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:

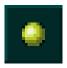
- `"team red <name>"`, where `<name>` is a name for the red team;
- `"team blue <name>"`, where `<name>` is a name for the blue team;
- `"score <redscore> <bluescore>"`, where `<redscore>` and `<bluescore>` are two integers representing the scores of the red and blue teams, respectively;
- `"credits <redcr> <bluecr>"`, where `<redcr>` and `<bluecr>` are two integers representing the credits of the red and blue teams;
- `"add <id> <image> <x> <y>"`, where `<id>` is a unique string identifier for an object (player, goalie, or ball) on the board, `<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.
- `"move <id> <x> <y>"` moves object whose identifier is `<id>` to the new location specified by `<x>` and `<y>`;
- `"change <id> <img>"` changes the image of object `<id>` to `<img>`;
- `"remove <id>"` removes object `<id>` from the board;
- `"chat red <text>"` displays the string `<text>` in the chat window. The string `<text>` can contain spaces, and must not be quoted.
- `"chat blue <text>"`, same as above, but for blue;
- `"reset"` resets the entire board.

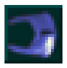
All of the above commands require `<x>` and `<y>` to be valid coordinates and `<img>` to be a valid image name. Also, `<id>` to be an identifier for an existing object, except for `add`, when it must be a fresh identifier. The list of possible image names for `<img>` are listed below. In this list, *i* is on of the directions 0 through 7:

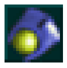
[1] This is done by invoking `NetGraphics.setup`, with a list of pairs (*host*, *port*) as arguments. For instance, `NetGraphics.setup [("localhost", 2005)]` connects to the port 2005 on your machine.

Image	**Description**
blue*i*	A blue robot looking in direction *i*
red*i*	A red robot looking in direction *i*
blue*i*b	A blue robot looking in direction *i* with the ball
red*i*b	A red robot looking in direction *i* with the ball
goalie*i*	A goalie robot looking in direction *i*
goalie*i*b	A goalie robot looking in direction *i* with the ball

Here are a few example of image names and their corresponding images:


ball

red4

blue4

red7b

blue5b

goalie2b

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

```
"team red Smashing Functors"
"add player2 red0b 4 7"
"change player2 red0"
"move player2 4 8"
"remove player2"
```

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 PS5.

6.2 Designing the world

Your first task is to implement the Multi-Ball world in the files `world/action.sml` and `world/game.sml`, and any files you choose to add. Note that you should only add files 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 we provide to you to test your world.

6.3 Designing a bot

Design a bot in the file `cl/mybot.cl`. This bot should be able to beat consistently the bots provided by the course staff. You will be graded on the number of times your bot beats our staff bots, and the strategy which it uses. We will make a server available for you soon, so you can try your bot against a bot developed by the course staff, and also make sure that your bot runs correctly on the server that we will use to grade it.

6.4 Documentation

As with all of the assignments up to this point, you should submit some documentation regarding your problem set. 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 any changes to or refinements of the specifications given in the problem set 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, particularly testing the bots, graphics, and world.

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 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.
- **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 bot. Start with the easier actions and work up to the harder ones. Actions like `turn` and `kick` are probably easier to implement relative to the other actions.
- **The ball can intercept the bots just as bots can intercept the ball.** It is obvious that if a bot moves into a tile where the ball is currently located, then the bot "catches" the ball. However, it is also important to remember that if the ball moves into a tile where there is currently a bot, the bot should then have control of the ball.

There are also many different strategies for building a good bot team. Here are some specific points to consider:

- **Your bots can communicate.** Your bots share a global memory that the other team cannot access. Use it to your advantage to coordinate your bots' movements.
- **You may have team members kick the ball to each other or carry the ball**. Kicking the ball makes it move faster than the opponent bots while carrying the ball give you more control if the enemy is nearby.
- **You may assign bots to specific regions of the board or all your bots may go for the ball.** Specialized bots allow you to assign defense and offense roles and spread them out on the board while swarms going for the ball makes the opponent work harder to avoid them.

6.6 Checkpoint submission

For this assignment, there will be a *checkpoint submission* halfway through the assignment. You are expected to submit: 1) a zip file `checkpoint.zip` containing your work at that point; and 2) a 1-2 page document `design.txt` (or `design.pdf`) that explains the design of your system. You will submit these files by April 27, at 11pm.

You are expected to have the design finished, and all the signature `.sig` files that you plan to add ready by that time. We will not look at the rest of the code, unless you have a poor final submission. In that case, if your submission reveals little work done by the checkpoint time, then the overall penalty will be more severe. On the other hand, if your final submission is well-documented, well-written, and runs without errors, then we will completely ignore your checkpoint submission code.

We strongly encourage that you to come discuss your design with the course staff during consulting/office hours.

6.7 Final submission

You will submit: 1) a zip file `ps6.zip` of all files in your `ps6` directory, including those you did not edit; and 2) your documentation file `doc.txt` (or `doc.pdf`). Although you will submit the entire `/ps6` 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 `/ps6` 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 `ps6` directory to compile your code without errors or warnings. *Note: Submissions that do not meet this criterion will be docked points.*

7 Tournament

At the end of the semester, there will be a competition between the robot programs of students who wish to compete. Each student project group may submit a robot program that will play against other student projects. 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.

`gfx/*`	Graphics files for the GUI
`world/action.sig`	Signature file for handling an action
`world/action.sml`	Functions for handling an action from a bot and implementing all local actions for locking memory
`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/*.java`	The java files that implement the GUI
`gui/*.class`	The GUI class files
`cl/*.ch`	Libraries for lists and booleans
`cl/sample.cl`	Sample team robot

9 Running the game

There are two main ways to run the game. One is running on your own world with the graphics connected to your local game. The other is running your bots on one of our servers, watching the game with the graphics connected to our game. You will need Java version 1.4.2 or later in order to run the graphics. 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.

9.1 Running a local world

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 `ps6/gui` directory and run `"java Gui <host> <port>"`. This tells the graphics program to start up and connect to the SML world running on `<host>` at port `<port>`. The `<host>` and `<port>` parts are optional. The default (if you run `"java Gui"`) is `localhost:2005`. Once the GUI is started, a board will pop up on your screen. The board will remain dark 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 `ps6` directory and run `CM.make()` to compile the program. Then run `"Game.start(<host-list>, <first-bot>, <second-bot>)"`, where `<host-list>` is a list of hosts (represented as hostname and port pairs), and `<first-bot>` and `<second-bot>` are strings representing file names you want to use as the two teams. Alternatively, you can run `T.test()`, which starts up a game between two sample bots on `localhost:2005`.

The game should now begin. When the SML program connects to the GUI, the playing board will light up. If at any point you need to recompile and start the game over, you will also need to restart the graphics program.

9.2 Our servers

To help you test your bots, even if your world is not complete, we will be providing several servers for you to connect to and run your CL code. Some of these servers will let you play against our own StaffBot while others will let you play against each other. The implementation of the world running on our servers will be our solution, so you will be able to see the kind of behavior we expect out of the world. *Note: Any attempt to break into these servers will result in all of the servers being taken down and full prosecution of the offenders.*