CS 312 Problem Set 6: λ Ball

Assigned: April 10, 2003 | Due: 11:59PM, May 2, 2003
Spec Change: April 19, 2003 | Design Reviews: April 16-19, 2003

1 Introduction

Welcome to λ Ball! You will be using the RCL interpreter you wrote in PS5 to implement a robot soccer game. This will be accomplished by implementing a new world and its actions. We have provided a graphic depiction of the game written in Java that can be used to monitor to progress of the game. You will keep the same partner you had for PS5. If exceptional circumstances arise, consult the course staff. **Warning: Some of the figures in this writeup could be hard to understand if you do not view them in color. Make sure you either read the Problem Set on your computer or print it out in color.**

1.1 Source code

The source code for the problem set is available as a zip file ps6.zip.

1.2 Clarifications and changes

The following clarifications or changes have been made to the problem set since it was released:

- April 13: Stun time on losing a challenge was made consistent.
- April 17: The timing of ASRs, status lists, and bot waiting was clarified in Section 5. Additionally, the notion of clock cycle was explained more thoroughly in Section 3.
- April 18: Fixed equation for the distance between two tiles to include absolute values.
- April 21: If a process halts or is terminated, any locks that it is holding should be automatically released.
- April 22:
  - Clarified number of spawn credits a team initially has in the spec change (a team starts with 9).
  - Goalie cannot leave goalie area; fixed in rules section
  - If a bot is near a wall and calls `do(A_LOCABS,x)`, if tiles are not within the bounds of the board, it returns `T_WALL` for those spots.
  - Fixed issue in spec change regarding the constant for an overloading bot with the ball—it is the same as the constants for overloading bots without the ball.
  - Fixed constant returned if a bot tries to spawn another bot illegally. It should have been `~1` and not `1`.
  - Clarified some of the requirements for the project pertaining to documentation in the Tasks section. You will need to turn in some documentation on this project. See Section 8.5 for more information.
- April 25:
  - Clarification on the definition of a free tile for purposes of spawning powerups and bots.
  - Fixed ASR for TURN in one of the examples.
  - Clarified LOCABS description.
  - Fixed description of new constants in spec change.

1.3 Use of RCL

You will implement your bots using the RCL language; they will run on your ps5 evaluator. You should not have to change your evaluator, except to fix any problems you had in it. To make RCL programming easier, we have added a number of language features that are already implemented for you in the new parser. These features are described in Section 6.

1.4 Specification change

A week after the problem set is handed out, there will be a specification change. This could be a change in the rules of the game, the layout of the board, the language, or features of the bots. The goal of the specification change is *not* to trick you or to make any code you have written irrelevant. If your code up to the point of the spec change is written well, you should not have to make large changes. The code you turn in will be required to implement the assignment as specified after the changes. However, *do not* wait to start programming until after the specification change—you won't have time.

2 The board

The board consists of 737 hexagonal tiles as shown in Figure 1. On each side of the board is a goal area. Only the goalies can enter these goal areas, which are shown by the lighter tiles on the board. The goal is immediately behind the goal areas.

2.1 Directions

Each tile has a six sides, numbered from 0 to 5 as shown in Figure 2. For robots on the team on the left, 0 is to the right. For robots on the right team, 0 is to the left. Thus, direction 0 always points towards the opposing goal.

2.2 Coordinate system

The board has an interesting coordinate system. Each location is described by a triple of integers (u, v, w) that sum to zero. The center of the board is at $(0, 0, 0)$. From a given tile on the board, moving in a direction increments one of the three values, decrements another, and leaves the final one the same, as shown in Figure 3. Appendix C shows the coordinates of every tile on the board.

The world gives the coordinates and directions to the bots in such a way that both teams think they are playing to score on the right side of the board. Hence, the coordinate a red bot sees as $(\tilde{}10, 11, \tilde{}1)$ appears as $(10, \tilde{}11, 1)$ to a bot on the blue team. Similarly, the direction 0 for the red team is the direction 3 for the blue team. This allows bots to be programmed without regard to the team they are on. Note that the tiles (x, y, z) and $(-x, -y, -z)$ are directly opposite each other on the board, so converting between one team's coordinate system and the other's is achieved by negating all the coordinates.

The *distance* between any two tiles is the smallest number of steps necessary to get from one tile to the other. If two tiles have coordinates (u_1, v_1, w_1) and (u_2, v_2, w_2), the distance between them is $(|u_2 - u_1| + |v_2 - v_1| + |w_2 - w_1|)/2$. Figure 4 shows the tiles located at various distances from a center tile.

3 Rules

As mentioned in the introduction, the rules of λ Ball are similar to those of soccer. What follows is a description of bots' actions, ball movement, starting, and scoring.

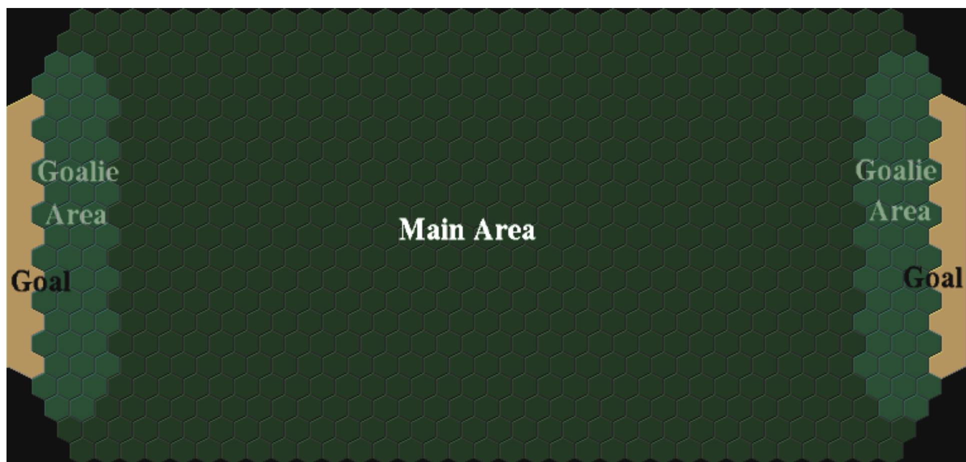


Figure 1: The λ-ball board

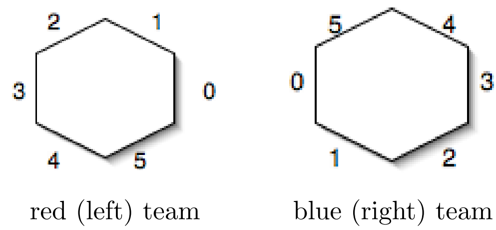


Figure 2: Sides of the hexagonal tiles

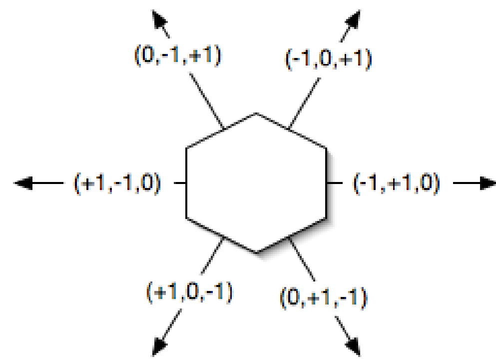


Figure 3: Coordinate changes from a tile

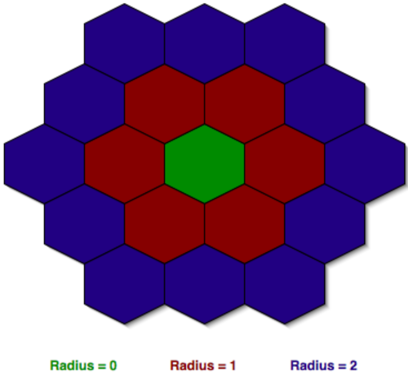


Figure 4: Distance from a center tile

3.1 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 other. The board also has a ball, which may be carried by bots or may be in motion. The ball has a location and a direction (0-5) in which it moves if it is not possessed by a bot.

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.

3.2 Starting the game

When the game starts, the ball is stationary at board coordinates $(0, 0, 0)$ with current direction 6. Each team is controlled by an RCL program. The first RCL process that starts for each team is registered as the goalie. The goalies start at position $(16, -16, 0)$. 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.

3.3 Spawning new bots

After the game start, both teams can spawn more bots using the `spawn` command in RCL. 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. The world returns to both bots the PID of the spawned bot.

3.4 Ball movement

The ball starts at coordinates (0, 0, 0) and stays stationary until a bot picks it up. Once picked up, the 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 three things happens:

1. **It comes in contact with another bot.** If the ball enters a tile where a bot currently sits, that bot captures the ball. The ball continues to move with that bot until the bot kicks it or another bot steals the ball. If a goalie controls the ball for more than 3000 clock cycles, the ball leaves its possession and returns to the center of the board.

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 $(15, -21, 6), (16, -21, 5), (16, -20, 4), \ldots, (21, -15, -6)$. 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 $(0, 0, 0)$) 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.

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., coordinates $(18, -7, -11)$, $(0, 11, -11)$, $(-15, 4, 11)$. When the ball runs into the wall, it bounces back in the tile from which it came in the direction next to the direction it entered that tile. The only tiles for which this is not true are the ones on the four angled sides of the board near the goals, e.g., $(11, -21, 10), (12, -21, 9), (21, -11, -10), (-11, 21, -10), (-14, 21, -7), (-21, 11, 10)$, and $(-21, 13, 8)$. In these corners, the ball bounces back in the direction from which it came. Possible bounces are:

When the ball is moving on its own, it should slow down gradually. When the ball is set in motion, it moves once every 20 clock cycles. After each move, this number increases by 15. Since robots take approximately 120 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.

3.5 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 the game status, the PIDs of the players on both teams, and a list of all the items within a certain radius. 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 the ball, $\frac{1}{4}$ if A has the ball, and 1 if B has the ball. The outcome of the fight depends on these probabilities and on whether or not one of the bots has the ball. If either of them had the ball, the winner of the fight gains possession of the ball. If B won the battle, then B stays in the tile and Bot A doesn’t move from the tile it was originally on and is instead stunned for 200 clock cycles. 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 in that tile. If the push is allowed, then A moves into the tile it was trying to and B is pushed into an adjacent tile. If the push is not allowed then both the Bots remain in their original tiles; the ball however, is possessed by the winner if either bot had it.
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 only fails if the direction passed in is invalid.
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 or if the bot is currently facing a wall. In the former case, the bot continues executing as it would have. In the latter case, the bot keeps the ball and continues to execute.

3.6 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 RCL code for the bots. The order of the evaluation must be fair to both teams.

Each of the bots as well as the game world should receive an equal number of clock cycles. The game world may use these periodic clock beats to move the ball. Since there are two teams each with their respective queue of bots, these queues should be given clock cycles in an alternating fashion until all the bots have been allocated exactly one step. The world is then given one clock cycle before the process repeats.

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.

4 Synchronizing processes

Some additional actions support synchronizing robot accesses to shared global memory. Consider a bot running this program:

```
let x = gref 0 in
  let inc = fn y => x := !x + y ; !x in
    spawn inc; spawn inc;
```

This code creates two bots with the ability to change the global reference `x`. What if these two processes execute in a way such that the first bot in the queue executes `!`ℓ `+` a, where ℓ is the location where the value of the reference is stored and a is some arbitrary value, and the next one is about to evaluate ℓ `:=` b, where b is some arbitrary value? When the second bot goes to update the reference, it will write the value $\ell + a$. If the reference is supposed to be keeping track of some information that both bots are to update, we'd more likely want the value to be $\ell + a + b$. Since the first bot retrieved the value of the reference before the second bot wrote it, the second bot's update is lost.

This is an example of a *race condition* : two processes are trying to access the same memory location, but there is nothing to *synchronize* when their accesses occur.

Locks are a good way to prevent race conditions. In order to access a location in the global memory, a bot must first *acquire* a lock. Once the lock is acquired, we say that the process *holds* the lock. Only one process may hold a given lock at a time. If a bot holds a lock, then any process that tries to acquire the lock blocks and does not execute until the lock is explicitly *released* by the process holding the lock. In general there may be several processes trying to acquire the lock at the time it is released. Only one of the waiting processes can acquire the lock and proceed; the others continue to block.

Here is the interface to locks:

Action	Call With	Description
lock	`do(A_LOCK,`ℓ`)`	Tries to acquire a lock. The global location ℓ is used as a way to name the lock. If the lock is not currently held by another process, the lock is acquired by this process and the process continues executing. If the lock is currently locked, the system returns an action ID until the location is unlocked, at which time one of the processes waiting on the location acquires the lock.
unlock	`do(A_UNLOCK,`ℓ`)`	Releases the lock named by ℓ. If the process does not hold the lock then the action has no effect.

Although it is convenient to use a location to name the lock, it is not strictly necessary—the locking interface could have used integers, for example, to name locks. It's simply convenient to name locks using global locations. In fact, a single lock may be used to synchronize accesses to many locations, not just the single location that is used to name the lock.

If a location or set of locations is meant to have only one process access it at a time, then an access to that location by *any* bot should be preceded by acquiring some lock and followed by releasing it. For example, the code above might be rewritten as follows:

```
let x = gref 0 in
  let inc = fn y => do(A_LOCK,x) ; x := !x + y ; do(A_UNLOCK,x) ; !x
  in
    spawn inc; spawn inc;
```

Now when the code is executed, the first bot to call the function will lock the location `x`. As a result, the second bot will wait to execute its code in the body of the function until the first bot has called `do(A_UNLOCK,x)`, and thus both updates to `x` will occur as intended.

If a process halts or is terminated, any locks that it is holding should be automatically released.

5 Implementing actions

The bots have a list of actions that they can take via the `do e` command. The time required to do an action depends on the size of the robot's team–smaller teams are quicker. Specifically, each action takes $t_b(1 + \alpha n)$ where n is the number of players on the bot's team, t_b is the base time for the action, and α is 0.1. Handling the time actions take is done through action IDs, as described in Problem Set 5. A bot that calls an action gets returned to it a new action ID $\lfloor basetime(1 + 0.1n)\rfloor$ times, and then the return value of the function.

Each action returns a pair containing a status list and an *Action-Specific Return* (ASR). The status list has the format [$ballradius, ballmajdir, ballmindir, adj, coord$]:

ballradius	the radius at which the ball is relative to the bot
ballmajdir and *ballmindir*	are the best two directions in which to move (one after the other) that will lead the bot closest to the ball from the current position, assuming the ball does not move. If the bot has the ball, both values are 0. If the ball is in a straight line from the bot, both values will be the same–the direction to the ball.
adj	is a list of the items in the six adjacent tiles, indexed by the direction. The possible return values are in Figure 5
coord	is the coordinates of the current bot

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. The status list, however, should be generated *after* the time the bot has to wait for the action to return.

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

Figure 5: Return values in the adjacency list

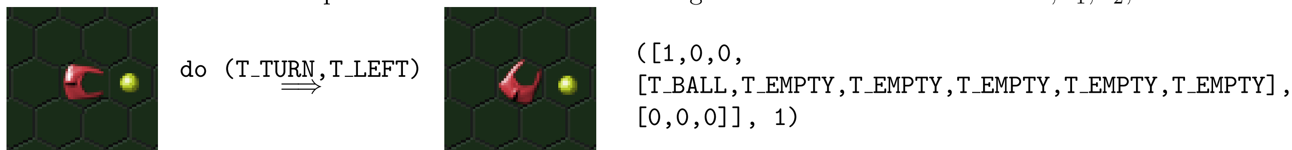
These and other constants are defined in rcl/constants.rch.

5.1 Possible actions

Figure 6 describes the possible bot actions in more detail.

5.2 Examples of actions

What follows are some graphical representations of what would happen and the return values given a bot calling certain actions. Assume for examples where the position of the bot is not explicit, assume bot is currently at the point $(0, 0, 0)$. Values that depend on position on the board, position of the ball, or robot PIDs where that information depends on factors not in the image are denoted with variables $r, d_1, d_2, \ldots$.

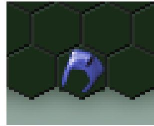


Command	Base Time	Args	Description	ASR	
A_MOVE	AT_MOVE	None	Move the bot forward	R_SUCCESS	if the move was successful
				R_HITWALL	if the bot tried to move into a place where it cannot
				R_WON	if the bot tried to move into a position with another bot who had the ball and won control of the ball (may 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 take an additional 200 clock cycles (a "stunned" period).
A_TURN	AT_TURN	Int i	Turn in direction i, where $i =$ T_LEFT or T_RIGHT.	d	the direction in which the bot is currently facing as described in Section 2 if the turn was successful
				R_FAIL	if the argument was not T_LEFT or T_RIGHT.
A_KICK	AT_KICK	None	Kick the ball forward	R_SUCCESS	if the kick was successful
				R_HITWALL	A wall was in front of the bot
				R_NOBALL	The bot did not have the ball
A_LOCABS	AT_LOCABS + n^2	Int n	Locate the items (and possibly their pids) within radius n.	$[[x_1, c_1, p_1], \ldots]$	where x_i is a the item type, c_i is the coordinates $(u_i, (v_i, (w_i, 0)))$ of the item, and p_i is is the PID if the item is a bot, the direction the item is traveling if it is the ball, and the constant PID_NONE otherwise. The list is ordered first by the radius of the items from the bot and then by the direction the items are in, starting at 0 and going counterclockwise. If there are tiles outside the boundaries of the board that are within the specified radius, return T_WALL for those tiles. The list of items for the ASR is generated *before* the wait time.
				R_FAIL	if $n < 1$
A_HEADCOUNT	AT_HEADCOUNT	None	Determines the PIDs of the teammates and opponents	(l_1, l_2)	where l_1 is the list of PIDs of the teammates and l_2 is a list of opponents' PIDs, both ordered by PIDs.
A_GETSTAT	AT_GETSTAT	None	Returns the score of the game	(s_1, s_2)	where s_1 is the score of the bot's team, s_2 is the score of the opposing team.
A_TALK	AT_TALK	String s	Prints s in the text window	R_OK	

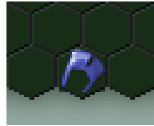
Figure 6: List of Actions

do T_MOVE $\Longrightarrow$

```
([0,0,0
[T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY],
[~1,1,0]],R_OK)
```



do T_MOVE $\Longrightarrow$



([r,d_1,d_2,
[T_EMPTY,T_WALL,T_WALL,T_EMPTY,T_EMPTY,T_EMPTY],
[u,v,10]],R_HITWALL)

do T_MOVE $\Longrightarrow$

([r,d_1,d_2,
[T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY],
[~1,1,0]],R_OK)

do T_KICK $\Longrightarrow$

```
([1,0,0
[T_OPPGOALA,T_EMPTY,T_EMPTY,T_EMPTY,T_EMPTY,T_OPPGOALA],
[~19,11,8]],R_OK)
```

do (T_LOOKABS,2) $\Longrightarrow$

([2, 0, 0,
[T_EMPTY, T_EMPTY, T_EMPTY, T_EMPTY, T_EMPTY, T_EMPTY],
[0,0,0]],
[[T_EMPTY, [~1,1,0], PID_NONE],
[T_EMPTY, [~1,0,1], PID_NONE],
[T_EMPTY, [0,~1,1], PID_NONE],
[T_EMPTY, [1,~1,0], PID_NONE],
[T_EMPTY, [1,0,~1], PID_NONE],
[T_EMPTY, [0,1,~1], PID_NONE],
[T_OPPWITHBALL, [~2,2,0], p_1],
[T_EMPTY, [~2,1,1], PID_NONE],
[T_EMPTY, [~2,0,2], PID_NONE],
[T_EMPTY, [~1,~1,2], PID_NONE],
[T_EMPTY, [0,~2,2], PID_NONE],
[T_EMPTY, [1,~2,1], PID_NONE],
[T_OPPONENT, [2,~2,0], p_2],
[T_EMPTY, [2,~1,~1], PID_NONE],
[T_TEAMMATE, [2,0,~2], p_3],
[T_EMPTY, [1,1,~2], PID_NONE],
[T_EMPTY, [0,2,~2], PID_NONE],
[T_EMPTY, [~1,2,~1], PID_NONE]])

6 RCL extensions

We've implemented some useful features in the RCL parser. These changes will make it easier for you to program bots. Note that none of these features should require you to change your evaluator; the new parser automatically *desugars* the new syntax into the appropriate AST.

6.1 Recursive functions

You can now declare recursive functions without having to use the "pass the function as an argument" trick. Simply use a `let` declaration. For instance, you could declare and use the factorial function as follows:

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

Note that you no longer need to use the `fn` keyword to declare functions. You can instead declare a named function with the SML-like syntax

`let` *name arguments* = *expr*

Functions declared in this manner may take multiple arguments, and may be recursive. The code above is translated into code like the following:

```
let fact = Y (fn fact => fn x => ifz x then 1
                                 else x*(fact (x-1))
in
    fact 3
```

where `Y` is the Y combinator, which will be covered later in the course.

6.2 Includes

You may wish to write code that multiple RCL 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 RCL file is located. If you execute SML from the ps6 directory and want to include in a bot the constants.rch file we have written–which is in the rcl directory, you would use

```
#include rcl/constants.rch
```

When this line is read, the file rcl/constants.rch 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.rch with `#include botfunctions.rch`, which contains

```
let trymove = fn x => ifz x 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`.

6.3 String parsing

The parser now handles strings, which is convenient for generating diagnostic output. Strings are written using double quotes; the parser automatically converts them to the string representation described in PS 5. For example, the parser converts the string `"abc"` to the expression `(97, (98, (99, 0)))`.

6.4 List library

We have provided a useful library for manipulating lists, located in the file rcl/lists.rch. The functions in the list library are similar to those in SML, as shown in Figure 7. 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 RCL as $(x_1, (x_2, (\ldots (x_n, 0) \ldots)))$. From now on, when we refer to a list in RCL (denoted as $[x_1, x_2, \ldots, x_n]$, we mean the corresponding RCL representation.

`cons x y`	Add the item `x` to the head of the list `y`
`head l`	Return the head of the list `l`
`tail l`	Return the tail of the list `l`
`length l`	Return the length of the list `l`
`nth l n`	Return the `n`th item 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 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`
`mergesort l`	Use a merge sort to sort the list `l`

Figure 7: RCL List Library Functions

7 Java GUI

Your program will talk to a Java program that graphically represents the game. This program provides a sprite-based interface that allows your program to update the display.

`JavaOutput.add(img:string,(u,v,w):(int*int*int)):JavaOutput.id`

Requires: `(u,v,w)` is a valid coordinate on the board, `img` is the name of a `png` file (without the extension) located in the `gfx` directory.

Effects: Adds the image `img` to the board at position `(u,v,w)` and returns the graphic ID of the newly-created sprite.

`JavaOutput.remove(gid:JavaOutput.id):unit`

Requires: `gid` is a valid graphic ID.

Effects: Removes the graphic corresponding to the given `gid`, which is made invalid.

`JavaOutput.change(gid:JavaOutput.id,img:string):unit`

Requires: `gid` is a valid graphic ID, `img` is the name of a png file (without the extension) located in the `gfx` directory.

Effects: Changes the graphic corresponding to the given `gid` to the new graphic. `img`

`JavaOutput.move(gid:JavaOutput.id,(u,v,w):(int*int*int)):unit`

Requires: `gid` is a valid graphic ID, `(u,v,w)` is a valid coordinate on the board

Effects: Moves the graphic corresponding to the `gid` to the new position `(u,v,w)`.

`JavaOutput.score(x:int,y:int):unit`

Requires: None

Effects: Displays the score of the red team as `(x)` and the score of the blue team as `(y)`.

The strings corresponding to the possible images are:

File	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

Figure 8: Examples of graphics in the game

8 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.

Additionally, you will be required to set up a design meeting with one of the TAs. These 20-minute meetings will take place a week after the problem set has been handed out, to determine how you plan to approach the project. Sign up early to get the time slot you want!

8.1 RCL interpreter

For the game to work, the RCL interpreter must be correct. We are not asking you to do any new implementation work on the RCL interpreter, but you are expected to fix any bugs in the interpreter that you submitted for PS5.

8.2 Designing the world

Your first task is to implement the λ Ball world in the files world/world.sml and world/action.sml. All of your changes for the world should occur in these files. Here you must implement the actions listed in Section 5. The local actions (for locking) should be implemented in action.sml and all of the functions listed in Figure 6 should be implemented in world.sml. You must also make sure that the actions bots take are rendered in the graphic display using the interface detailed in Section 7. You can use the bots we provide to you to test your world.

8.2.1 Things to keep in mind

This part of the program promises to be challenging. There are many factors to consider while designing the world. Here are some issues to keep in mind as you write your code.

- **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 a good idea to use the bots we provide you for testing, because you can be sure they do what they are supposed to given that the world is implemented correctly.
- **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.

8.3 Designing a bot

Design a bot in the file rcl/mybot.rcl. 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 later make a server available for you so you can try your bot against a bot developed by the course staff, and also so you can make sure that your bot runs correctly on the server that we will use to grade it.

8.3.1 Things to keep in mind

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

- **You can have either a few faster bots or many slower bots.** Both ways have advantages and disadvantages. While faster bots could react more quickly and win more "fights" due to the small team size, the slower bots would have an advantage in numbers and would be able to cover the board well.
- **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.

8.4 Design meeting

Design meetings will take place between you, your partner, and a TA on April 16–19. You will sign up for a twenty-minute meeting in section on Monday, April 14th or after that point using a signup sheet outside Professor Myers' office (Upson Hall 4119C). The meetings will be available at a variety of times during the day, so you should be able to find one that suits your schedule. During this meeting, you will discuss with the TA your approach for implementing the project. This includes breakdown of the program, issues you foresee needing to overcome, etc.

Before the meeting, you should prepare a 2-page document that includes your interfaces and a description of your design. Module dependency diagrams are recommended. Email the design document to the TA you are going to see no later than **11:59pm the day before your review**. The format of the document should be plain text or PDF. This way, the TA will be familiar with your ideas before the meeting and therefore the time spent will be more productive and helpful for you.

Your design meeting presentation will be worth 10% of the total grade for this assignment, so make sure to prepare for it properly.

8.5 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 for the design review. 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.

8.6 Files to submit

You will submit a zip file of all of the files in the `ps6` directory, including ones you edited and ones you did not edit. You should not need to add any new files. New structures or signatures that you define can be included in one of the existing files.

9 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. The winning project group will receive a prize. Details on the tournament time and location and the submission procedure will be available later.

A What You Are Given

A.1 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 the files specifically listed in Section 8.6. 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-util.sml	Handles network connections for the game server
world/game.sml	Handles starting a game and talking to the network
world/world.sig	Signature file for the functions contained in the world
world/world.sml	All of the functions for the Sλ Ball world.
world/world-util.sml	Functions to help you in implementing the world.
io/Error.sml	Error handler for IO functions
io/GameIO.class	Graphics program (written in Java) for the game that receives messages from SML
io/GraphicInterface.java	Handles the drawing of graphics on the board
io/HexBoard.java	Creates the game board
io/JavaOutput.sml	Provides an interface between SML and Java.
io/Network.sml	Network functions needed for talking to Java
io/SocketUtil.sml	Socket functions for SML
io/poll.sml	Network stuff for SML
parser/*	Files for the new and improved RCL parser
rcl/*	Sample bots you can run in your world. Open the bots in a text editor and read the comments to see how they each act.

A.2 Functions

A number of functions are provided for you in the world already to make programming easier for you. The functions are all in world/world-util.sml. The implementations of these functions may be useful to you in figuring out how to work with the board's coordinate system.

`inBounds(u:int,v:int,w:int):bool`
Requires: None
Returns true if the given coordinates are within the bounds of the board, false otherwise

`isGoalArea(u:int,v:int,w:int):bool`
Requires: None
Returns true if the given coordinates are within the goalie area on either side of the board and false otherwise

`isInGoal(u:int,v:int,w:int):bool`
Requires: None
Returns true if the given coordinates are within the goal on either side of the board, false otherwise

B 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. Note that you will need the Java Runtime Engine, version 1.3+ in order to run the graphics. If you do not have it, you can download it from http://java.sun.com/getjava/manual.html. 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.

B.1 Running a local world

These are the steps you'd take to run a game on the local machine.

1. 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`.
2. At the command prompt, go the the ps6/io directory.
3. Run java GameIO *server port*. This tells the graphics server to start up and connect to the SML world running on *server* on port *port*. If you want to connect to the SML world running on your own machine, use java GameIO localhost 2001.
4. In another command prompt, go to the ps6 directory.
5. Make sure sml is installed in the default location. If not, change the file run.bat (run.sh on *nix systems) to reflect the location of sml.
6. From the command prompt (dos or *nix shell) execute run bot1 bot2. This starts up sml, runs CM.make(); and starts a game between bots located in rcl/bot1.rcl and rcl/bot2.rcl

Alternate method

1. Follow steps 1–3 above.
2. Start SML at a new command prompt, or in Emacs.
3. Run CM.make() to compile the program.
4. Run Game.start(*first bot*,*second bot*), where the bots are strings of the file names you want to use as the two teams. These will most likely be rcl/*something*.rcl.

The game should now begin. If at any point you need to recompile, you will need to restart SML. Otherwise, you will get an error message stating that the system cannot bind to the socket. Feel free to take a look in game.sml for other ways to start games.

To control the speed of the game, after calling CM.make() at the SML prompt, call Game.setGameSpeed(x) where x is a real number in seconds to wait between two clock cycles. The recommended setting is $x = 0.001$ for most computers. The setting needs to be set every time CM.make() is executed. Optionally, the default setting may be changed in Game.sml.

You may start up the graphics client once and repeatedly start and stop sml. If however you call CM.make() multiple times in the same sml session, you will need to restart the graphics client.

B.2 Our servers

To help you test your bots, even if your world is not complete, we are providing several servers for you to connect to and run your RCL 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. As the project progresses, more of these servers will have our StaffBot as the opponent so you can test to see if your bots can beat ours.

Servers will be available about a week or so after the project is handed out. The hostnames for the servers, ports on which they will be running and the bots they will be running are yet to be determined. However, a list of the servers running and their status will be available at http://www.dividedsoul.net/cs312/. On that site will also be directions for running your bots on our server and connecting your Java graphics client to the world running. **NOTE: Any attempt to break into these servers will result in all of the servers being taken down and full prosecution of the offenders.**

C Coordinates of all points on board