

# CS 312 Problem Set 6: $\lambda$ -Shark (CTF)

Assigned: April 15, 2004

Due: 11:59PM, May 6, 2004  
Design review: April 26–27, 2004

---

Virtucon Corporation has discovered that the originally planned  $\lambda$ -Shark game doesn't test well with focus groups, who complain that the game lacks variety and places too much emphasis on combat. They've asked you to instead develop what was meant to be a capture-the-flag expansion pack for  $\lambda$ -Shark. The primary object of the game is now to capture the opposing team's flag as many times as possible. In the writeup below, new elements in the specification are highlighted in blue; some text from the original spec is also struck out.

## 1 Introduction

A large multinational corporation, Virtucon, has hired your project group to use the RCL interpreter you wrote in PS5 to build a robotic ~~battle~~ [capture-the-flag](#) game called  $\lambda$ -Shark<sup>1</sup>. In terms of RCL, this will be accomplished by implementing a new world and its actions. 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.

### 1.1 Source code

Source code for this project is available in CMS.

### 1.2 Clarifications and changes

Watch this space for clarifications and changes.

### 1.3 Use of RCL

The game will be played by bots driven by programs written in the RCL 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.

[\(Original sections on the spec change and design review omitted here\)](#)

---

<sup>1</sup>In honor of the lambda calculus, a predecessor to ML and other functional programming languages

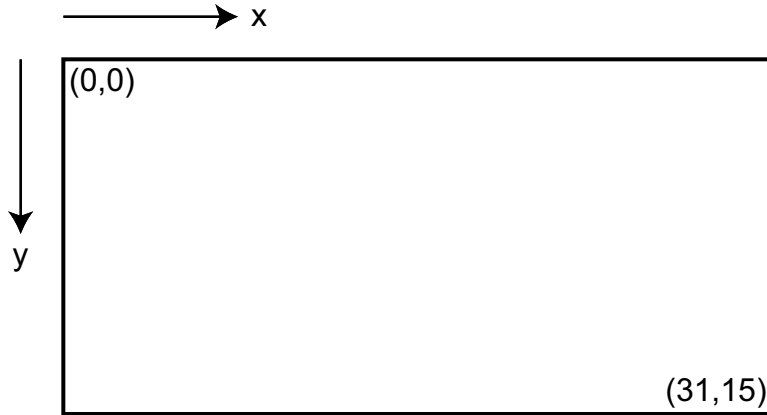


Figure 1: The board layout

## 2 Game Rules

$\lambda$ -Shark is a game played by two teams of RCL robots, the red team and the blue team. The object of the game is for your team to survive while destroying as many robots of the other team as possible. The team with the most surviving robots at the end of the game is the winner. The game is a version of capture-the-flag. The object of the game is to capture the opposing team's flag as many times as possible while defending one's own flag. A flag is captured when a robot on the opposing team picks it up and carries it all the way back to its own flag. The game runs for a maximum of 10,000,000 execution steps (about 7 minutes).

### 2.1 The Board

The  $\lambda$ -Shark board is a rectangle made of square tiles, as illustrated in Figure 1. The board is 32 tiles wide by 16 tiles tall. Board locations are named using Cartesian coordinates. The square in the upper left corner is (0,0), and the bottom right corner is (31, 15). Thus, positive  $x$  is right, and the positive  $y$  direction is **down**. The board does not wrap; for example, you may not travel left from board position (0, 6). Figure 2 illustrates the four directions in which a bot can move from the tile that it is on.

Tiles may be occupied by at most one object at a time. These objects are bots, powerups, [flag stands](#) and walls. Invalid board locations (that is,  $(x, y)$  where  $x < 0$ ,  $x > 31$ ,  $y < 0$ , or  $y > 15$ ) are treated as walls.

As described in Sections 2.9 and 2.10, powerups and new bots will appear on the board from time to time. The location at which these things can occur, as well as the locations of walls, [and flag stands](#), are attributes of the current map that is being used for the game.

### 2.2 Time

Time in the game is measured in steps. In one game step, each running bot (that is, each RCL thread) is allowed to take one execution step. The game lasts for at most 10,000,000 steps total. Some bot actions take more than one execution step: for example, moving, turning, and firing

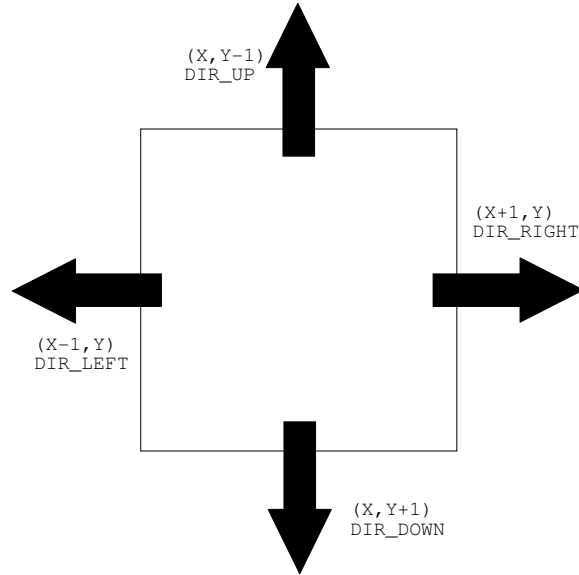


Figure 2: Board direction and coordinate conventions.


4	3	2	3
3	2	1	2
2	1		1
3	2	1	2

Figure 3: Manhattan distance from a tile.

the bot's weapon. The delay involved in these actions is implemented using action identifiers as described later.

### 2.3 Distance

Distance in  $\lambda$ -Shark is measured using *Manhattan distance*: the distance a taxi would have to drive in Manhattan. The distance between  $(x, y)$  and  $(x', y')$  is defined to be  $|x - x'| + |y - y'|$ . Manhattan distance is illustrated in Figure 3.

### 2.4 Scoring

The team with the largest number of robots at the end of the game wins the game. However, if all the robots of one team are destroyed, the other team wins immediately. Robot destruction can be

result of damage (see Section 2.12) or program termination either normally or by getting stuck in evaluation (self destruction). Ordinarily, the team with the largest number of points at the end of the game wins. Points are scored by capturing the enemy team's flag and bringing it back to one's own flag stand. A team may score whether or not its own flag is at its stand.

## 2.5 Game End

Games end after 10,000,000 evaluation steps or when one team's robots are all destroyed, or when one team scores their tenth point. The team with more points wins. If the teams have equal scores, then the game is a draw.

## 2.6 Health

Bots have a certain number of health points. They start with 3 health points, but successful attacks against a bot deplete that health. When health reaches 0, the bot is destroyed (see Sections 2.11 and 2.12). Bots may have additional state indicating whether they are carrying a flag.

## 2.7 Motion

At any given time a bot is located on some tile  $(x, y)$  and is facing in one of the four directions shown in Figure 2. A bot may turn to face in any of the four directions with a single action. Additionally, a bot may attempt to move in the current direction. If the tile adjacent to the bot in the direction it is facing is empty, the bot moves into the adjacent tile. If the adjacent tile is not empty, the robot does not move, and the rules given in Sections 2.8, 2.9, and 2.12 govern behavior. Both moving and turning take some time. The actual action takes place immediately, but the bot then waits for a number of steps before it can perform any more actions or computation.

## 2.8 Flags

A flag stand may be either occupied by a flag or empty. At the beginning of the game, each team's flag is located in the team's flag stand. When a bot attempts to move into the tile of the opposing flag stand, the bot does not change position. If the flag stand is occupied, the bot acquires the flag and the flag stand becomes empty. Once a bot has a flag, it is considered to have the flag until the bot is destroyed or attempts to move onto its own team's flag stand. In the latter case, a point is scored (see Section 2.4). In either case, the flag returns to its home flag stand.

## 2.9 Powerups

Robots may pick up spawn credit powerups that appear randomly. A powerup is picked up if a robot moves into the square containing it. Each team holds a shared reserve of spawn credits. When a bot picks up a powerup, its team's spawn credit count is incremented by 1. The powerup no longer exists once a bot occupies its square.

Powerups appear on the board at random intervals. The constant `ITEM_DELAY`, defined in `util.sml`, specifies the mean number of steps between powerup creations; a powerup may appear after each evaluation step with probability  $1/\text{ITEM\_DELAY}$ . When a powerup appears, it is placed at a randomly selected powerup drop location that is currently empty (see Section 2.1). If no drop location is empty then no new powerup is created.

## 2.10 Bot creation

When a bot executes a `spawn e` expression, a new bot may be created. However, the success of the spawn is dependent on the state of the game. A spawn is successful if there is an open spawn point of the appropriate color on the board and the spawning team has a sufficient number of spawn credits. In the event of a successful spawn an empty spawn point is chosen randomly and a new bot running `e pidparent` is added at that board location. The number of spawn credits required to spawn is as follows:

$$\text{spawn cost}(\text{team size}) = \begin{cases} 1 & \text{team size} < 5 \\ \text{team size} - 4 & \text{otherwise} \end{cases}$$

For example, a team of size 4 needs only one credit to spawn a new bot, but a team of size 7 needs 3 credits to spawn. The function `GameUtil.spawnCost` implements this. In the event of a successful spawn, the corresponding spawn credits are deducted from the spawning team.

If a spawn is unsuccessful (either because all the team's spawn points are occupied or because the team lacks sufficient spawn credits), no new bot is created and `spawn e` evaluates to 0. The team's spawn credits are unchanged.

## 2.11 Bot destruction

A bot is destroyed (“unspawned”) if its program terminates either by evaluating to a value or attempting to perform an illegal operation (that is, evaluation becomes stuck). It is also destroyed if its health is reduced to 0 or less. When a bot is destroyed, it is removed from the board and its tile becomes empty. Additionally, this bot must not be permitted to take any more evaluator steps. [If a bot is destroyed while carrying a flag, the flag is returned to its flag stand.](#)

## 2.12 Combat

Bots have the ability to fight each other using either close-range or long-range attacks.

### 2.12.1 Melee

When a bot attempts to move into a square occupied by another bot, the moving bot is the attacker and the stationary bot the defender. If the defender is on the opposing team, it loses 2 health; otherwise, the move has no effect. The attacker always waits after the attempted move, just as it would if the destination square had been empty.

If the attack destroys the defender, the defender should be removed as described in Section 2.11. The attacker does not move even if the defender is destroyed.

### 2.12.2 Laser Beams

Bots have lasers which may be fired in the direction that it is facing. The laser will hit the first non-empty tile in its path, including enemy robots, walls, powerups, [flag stands](#), etc. If the object it strikes is a robot, that robot's health is reduced by 1 point, even if it is a member of the same team. In any other case, the laser has no effect. Like moving and turning, firing the laser takes time.

### 2.13 Game Starting Conditions

Each team begins the game with 3 spawn credits and 1 bot, which is randomly placed on one of the team's spawn points.

## 3 Implementing the game

### 3.1 Map Files

Map files are defined by plain text files consisting of 16 lines of 32 characters. Each character is one of `{., *, r, b, R, B, i}` and has the meaning specified below.

- `.` empty space
- `*` wall
- `r` red spawn point
- `b` blue spawn point
- `R` [red flag stand](#)
- `B` [blue flag stand](#)
- `i` powerup drop point

The function `GameUtil.loadMap : string → mapdef` provides a method to read map files. You will want to use this function to build a better map representation.

### 3.2 Actions

Bots can perform a variety of actions using RCL do expressions. The identifiers for the actions are defined in `constants.rch`. Actions are named `A_x` where `x` indicates the kind of action. Some actions cause bots to be delayed by some number of steps. The delay corresponding to action `A_x` is named `AT_x`, and its value is given in `world/definitions.sml`.

Available actions are summarized in Table 1. *Objects* and *Dirs* are defined in Sections 3.3.2 and 3.3.1, respectively. Note that the lists described as result values are actually implemented as in the list library from PS5.

Action delays are implemented using action identifiers as described in the PS5 language description. When an action needs to cause the bot to delay, the world returns a new action identifier that will cause the bot to query the world again on the next execution step. Once the bot is to be permitted to continue evaluation, the world returns some other expression to be evaluated.

Command	Args	Description	Return
A_MOVE		The robot attempts to move. See Section 2.7.	R_OK if the bot moved; R_FAIL otherwise
A_FACE	$i \in Dirs$	The robot turns to face direction $i$ .	R_OK
A_SHOOT		The robot shoots a laser beam. See Section 2.12.2.	R_OK
A_INSPECT	$(x, y)$	Returns the object type located at board position $(x, y)$	code $c$ where $c \in Objects$
A_LOOK	$i \in Dirs$	From the bot's location, finds the first non-empty square in direction $i$	$(d, o)$ where $d$ is the distance to $o \in Objects$ .
A_NEAREST	$o \in Objects$	Returns the coordinates of the object described by $o$ that is nearest to the bot. Ties are broken arbitrarily.	coordinates of the object, $(x, y)$ , or R_FAIL if no object found.
A_MYSTATS		Returns the bot's current status	<del><math>[x, y, h]</math> where <math>(x, y)</math> is the bot's position, <math>h</math> is its health</del> $[x, y, h, f, d]$ where $(x, y)$ is the bot's position, $h$ is its health, $f$ is true if the bot has the flag, and $d$ is its direction.
A_TEAMSTATS		Returns team status	$[c, p_m, p_o, s_m, s_o, l]$ where $c$ is number of spawn credits the bot's team has, $p_m$ is the players on the bot's team, and $s_m$ is the score of the bot's team. $p_o$ and $s_o$ are opponents team size and score. $l$ is a list of the bot's teammates' pids.
A_FLAGPOS		Returns current flag positions. Note this can change when a flag is being carried.	$[x_m, y_m, x_o, y_o]$ where the bot's team's flag is located at $(x_m, y_m)$ and the opponent's flag is located at $(x_o, y_o)$ .
A_TALK	$s = [c_1, c_2, \dots]$	Sends message $s$ to the command line.	R_OK

Table 1: Table of game actions.

Code	Meaning
<code>O_EMPTY</code>	Tile is not occupied.
<code>O_WALL</code>	Tile contains a wall. Please note that locations not on the board are considered to be walls (see Section 2.1).
<code>O_MYFLAG</code>	Tile contains the team's flag.
<code>O_OPPFLAG</code>	Tile contains the opponent's flag.
<code>O_SPAWN</code>	Tile contains the spawn powerup.
<code>O_TEAMMATE</code>	Tile contains a teammate.
<code>O_TEAMMATE_WITH_FLAG</code>	Tile contains a teammate carrying a flag.
<code>O_OPPONENT</code>	Tile contains an opponent.
<code>O_OPPONENT_WITH_FLAG</code>	Tile contains an opponent carrying a flag.
<code>O_FLAGSTAND</code>	Tile contains a flag stand.

Table 2: Object code definitions.

### 3.3 Constants

#### 3.3.1 Direction Codes

The direction codes are `DIR_UP`, `DIR_DOWN`, `DIR_LEFT`, and `DIR_RIGHT`. and are explained in Section 2.1. The set of all direction codes is referred to as *Dirs*.

#### 3.3.2 Object Codes

Object codes are used to indicate the status of board square. They are returned by `A_LOOK` and `A_INSPECT` and used as input to `A_NEAREST`. Reference to “team” and “opponent” are relative to the bot performing the action. The set of object codes is referred to as *Objects*, and each individual code is defined in Table 2.

It is possible for a single square to match multiple object codes. While this does not affect `A_NEAREST`, we need to exercise caution specifying the output of `A_LOOK` and `A_INSPECT`. These functions return the most specific code that applies to the target square. The following table summarizes several corner cases. Behavior is symmetric with respect to team membership.

Square Contains	Output Code
Team flag and flag stand	<code>O_MYFLAG</code>
Teammate with flag	<code>O_TEAMMATE_WITH_FLAG</code>

## 4 Graphics

Unfortunately, interfaces to graphical functions from SML leave much to be desired, so instead the graphics will be displayed by a Java application that we have given you. SML connects to this program through TCP/IP and sends primitive commands to display images on the screen.



SML doesn't have built-in graphics support, so we have provided a simple graphics module. The Graphics module in `network/graphics.sml` provides the functions listed below.

```
setup(): unit
  (* Effects: establishes a connection to the graphical client.
     All other Graphics functions require that a connection exists. *)
reportMap(m: GameUtil.mapdef): unit
  (* Effects: Draws the map m.
     Requires: m is a valid mapdef generated by GameUtil.loadMap. *)
draw(s: string, x: int, y: int): unit
  (* Effects: Draws the image corresponding to s at (x,y).
     Requires: s is the name of an image. *)
erase(x: int, y: int): unit
  (* Effects: Draws an empty tile at (x,y). *)
reportShot((x: int, y: int), (x': int, y':int)): unit
  (* Effects: Draws a laser shot from (x,y) to (x',y').
     Requires: x=x' or y=y' *)
reportSpawn(r: int, b: int): unit
  (* Effects: Makes the spawn credit counter read r and b
     for red and blue respectively. *)
reportNames(r: string, b: string): unit
  (* Effects: Sets the displayed names of the red and blue teams to
     r and b respectively. *)
reportScore(r: int, b: int): unit
  (* Effects: Sets the displayed scores for the red and blue teams
     to r and b, respectively. *)
reportSpawn(r: int, b: int): unit
  (* Effects: Sets the displayed spawn credit counter to r and b
     for red and blue respectively *)
reportWin(s: string): unit
  (* Effects: Displays a message saying s is the winning team. *)
reportTime(t: string): unit
  (* Effects: Sets the displayed time to t. *)
report(s: string): unit
  (* Effects: Sends the string s directly to the Java client program.
     Do not call this function directly unless you really know what you are doing. *)
```

#### 4.1 Images

The function `Graphics.draw` takes a string corresponding to image as part of its argument. A few sample images are shown in Table 4, and the set of defined strings is given in Table 3.

## 4.2 Starting the graphics

The graphics library is actually implemented by a Java client program. SML connects to this program through TCP/IP and sends primitive commands to display images on the screen.

To compile the graphics client from the command line, type: `javac *.java` from the `gfx` directory. If you have difficulty getting the application to compile and run, you may need to download the latest version of the Java SDK (1.4) from <http://java.sun.com/j2se/1.4.2/download.html>.

When you run your game, you will need to separately start this client. To start it from the command line, type: `java` Start from the `gfx` directory.

## 5 Your Tasks

This project has several parts. You are advised to spend time thinking about each part before you start. Start on this project *early*. There are many things you will have to take into consideration when designing the code for each section. Starting with a good design can prevent many problems from occurring in the future. You will be required to set up a 20-minute design review with one of the TAs. These meetings will take place roughly a week after the problem set has been released, and will cover your design decisions and your method of approaching the project.

### 5.1 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 that may have been found in it from PS5. If you need assistance with this, come to office hours or consulting hours and we will be happy to help you.

### 5.2 World Design

You should think about implementing the world before you start coding. **Your code is required to enforce all of the rules.** In particular, the following are important things to think about:

- Robot movement according to the movement rules.

"RED_LEFT"	"BLUE_LEFT"	"WALL"
"RED_RIGHT"	"BLUE_RIGHT"	"POWERUP_SPAWN"
"RED_UP"	"BLUE_UP"	
"RED_DOWN"	"BLUE_DOWN"	
"RED_LEFT_FLAG"	"BLUE_LEFT_FLAG"	
"RED_RIGHT_FLAG"	"BLUE_RIGHT_FLAG"	
"RED_UP_FLAG"	"BLUE_UP_FLAG"	
"RED_DOWN_FLAG"	"BLUE_DOWN_FLAG"	

Table 3: Defined image names



Table 4: Sample images.

- Spawn credit powerups
- Spawning and unspawning
- Scoring [and handling the flag](#)
- Robot combat
- Execution step limit
- Graphics and interfacing with the GUI in JAVA

A good implementation is modular, clean, and adaptable. You should consider things that could be changed in the project and think about how your code would evolve.

### 5.3 Design Review

You and your partner will be required to set up a 20-minute design meeting with a TA about a week after the problem set is released. These meetings will be held at various times during the day so you should be able to find one that fits your schedule. During this meeting you will be expected to explain your entire design to the TA, as well as discuss possible issues with your implementation. This is a presentation and you should come prepared to lead the discussion of your design.

You are required to bring a document listing each of the modules you plan to use, with *fully specified signatures*. You should also describe in detail the representation of data in each module, and the data structures you plan to use. It is highly recommended that you bring a module dependency diagram as well. We expect both partners to have worked together in designing the modules. Therefore, each partner is responsible for having a complete understanding of the project and your proposed implementation.

E-mail all of your documents to the TA that you are going to see by midnight the night before your review. Your documents should be in PDF or plain text format. This way, the TA will be familiar with your design ideas before the meeting and can be more helpful during the meeting. Your design meeting will be worth 15 percent of your total project grade; make sure to prepare for it adequately.

You are not required to have a bot designed in time for the design review.

### 5.4 World Implementation

Implement the world according to your design. You are being given considerable freedom in your design, and should be able to responsibly design and implement a project of this scale. Remember to start early, and think before you code.

Your world needs to implement the rules explained in this document, and interface with the given Java client to produce watchable games.

## 5.5 Bot Programming

In addition to implementing the world, you need to program a bot in RCL capable of playing  $\lambda$ -Shark. To help you with this, we have provided you with `botlib.rch`, which contains some useful primitives for interacting with the world. Be sure to review this file as the functions it contains will illustrate how RCL programs interact with world. Your robot should be able to consistently beat the provided `random.rcl` bot. After the design reviews, we will bring several  $\lambda$ -Shark servers online. Your bots should also be able to beat `stupidbot`, which will be playable on the servers only.

These servers will likely be maintained on the `thunderbunny.net` domain. Do not attempt to attack these servers. You have been warned. These servers are provided for your benefit, and any abuse will result in the servers being taken down and action under the Cornell computer abuse policy <http://www.cit.cornell.edu/computer/policies/abuse.html>.

## 5.6 Karma Exercise

You may wish to explore the Java-based graphics client. The team that best enhances this system by adding functionality and/or new graphics will receive a prize of “refreshments” from the course staff.

Extending the game to allow for bots programmed in SML is an interesting, and challenging, extension.

## 5.7 Deliverables

You will submit a single zip file containing your source code and documentation for ps6. We are not specifying files to submit, so you are free to create new source files as necessary. Likewise your code will not be tested using automated tools that are dependent on your signatures. As this implies, you may add or change signatures to create well structured modular code.

Your zip file must contain a file called `readme.pdf` or `readme.txt`. This document should discuss your design, implementation, and testing strategies. If setting up your program is not trivial, please include directions. Your zip file must also contain a bot called `submission.rcl`, which we will test against `random` and `stupidbot`.

It is, of course, imperative that your code compiles.

## 6 Tournament

At the end of this semester there will be a competition between robot AI submissions. Participation is entirely voluntary, and students are encouraged to come to the tournament even if they do not wish to enter their bots. There will be free food. Each student group project group may submit

an entry and the winning team will receive a prize. Tournament time, location, and submission procedure will be announced in the 312 website and on the newsgroup.

## 7 Files

This program contains the following files and directories.

<code>gfx/*.java</code>	GUI server source code
<code>gfx/gui_status.png</code>	Status area background
<code>gfx/water_tiles.png</code>	Tile images
<code>gfx/water_backdrop.jpg</code>	Background image
<code>maps/*</code>	Sample map files
<code>network/client.sml</code>	Networking primitives
<code>network/graphics.sml</code>	Interface to graphics server
<code>rcl/random.rcl</code>	A bot that moves randomly
<code>rcl/constants.rch</code>	Constants for rcl programs
<code>world/definitions.sml</code>	Game constants in SML
<code>world/sigs.sig</code>	Signatures
<code>world/action.sig</code>	Action signature
<code>world/action.sml</code>	Action implementation
<code>world/game.sml</code>	Game logic
<code>world/loop.sml</code>	Main game loop– drives all other code
<code>world/util.sml</code>	Utility functions for loading maps, etc . . . .
<code>absyn/*</code>	Same as PS5
<code>compat/*</code>	Same as PS5
<code>debug/*</code>	Same as PS5
<code>eval/*</code>	Same as PS5
<code>parser/*</code>	Same as PS5