CS 312 Problem Set 6 (Project Part II): Twenty Thousand λs Under the Sea

Assigned: April 12, 2007

Final submission due: 11:59PM, May 3, 2007
Checkpoint meetings: April 19, 20 and 22, 2007

1 Introduction

In the previous part of the project (Problem Set 5), you developed an interpreter for a concurrent programming language. This part will allow you to put that language to good use: you will develop a game called *Twenty Thousand λs Under the Sea*, in which two underwater forces led by the gods Neptune and Poseidon compete to collect treasure. You will implement the world for this game (in SML), as well as the code for at least one game player (in RCL). We have provided some graphical support that you can use to display the progress of the game. You will keep the same partner you had for part I; consult Professor Myers if this is truly problematic.

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

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

Source code for getting started on this project is available in CMS.

1.1 Updates to problem set

- Treasures spawn uniformly at random in rows 11-15 in an EMPTY cell.
- When a squid or whale is immobilized by ink, that process expression is wrapped in a delay expression. In other words, the process will not proceed with its computation until after the delay.
- The point break down for ps6 is as follows:
  - Design meeting - 5pts
  - World - 60 pts
  - RCL team (AI) - 15 pts
  - Documentation - 5 pts
  - Complexity written problem - 13 pts
  - CVS log - 2 pts
- For the written problem: $m(n)$ is always the maximum size of the recent list, not the current size. Also, the resize happens when the recent list hits size $m(n)$, not $m(n) + 1$.

1.2 Use of RCL

The behavior of each team in the game is driven by a program written in the RCL language. You will implement not only the game but also a team to play the game. Your evaluator from part I will run this program. You will need to copy your part I code into the part II distribution in order to compile it. You should not have to change your evaluator code except perhaps to fix bugs.

2 Game rules

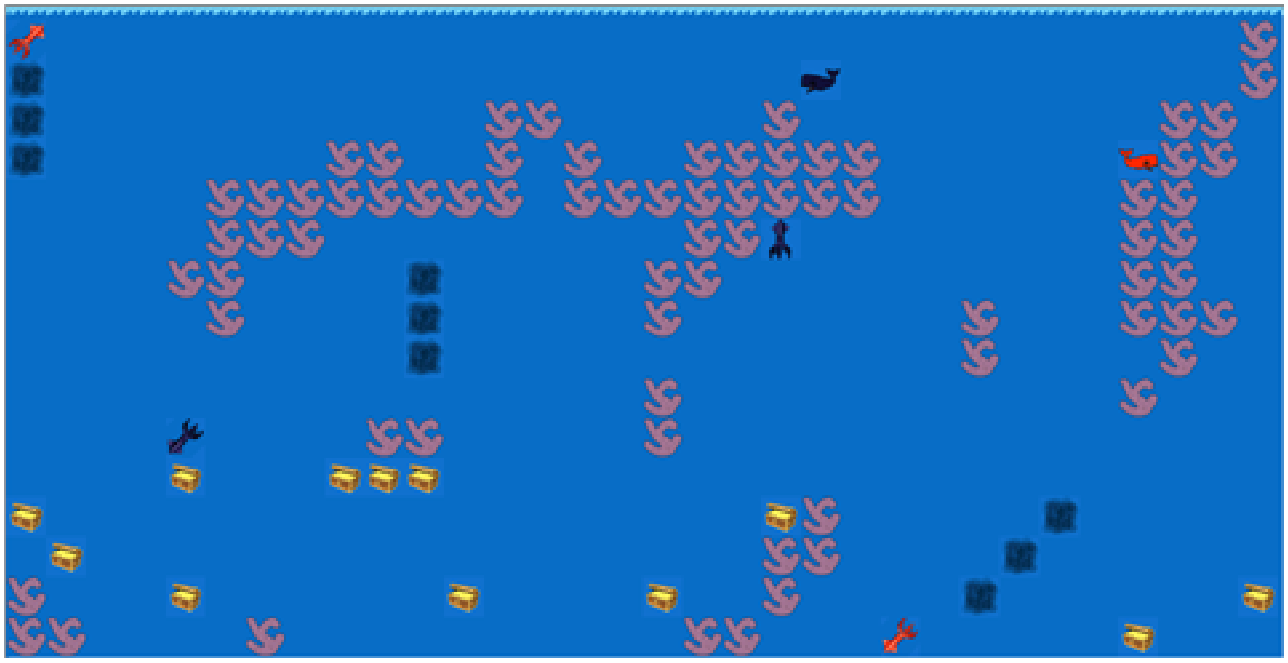
Twenty Thousand λs Under the Sea is a game played by two teams, the red team and the blue team. The object of the game is to maximize your team's treasure points before the time runs out. There are three kinds of bots on a team: gods, squids and whales. Each bot is driven by a single RCL thread. Because they are RCL threads, they can communicate with each other using global memory.

The red god is named Neptune and the blue one, Poseidon. Gods are not physically present in the game, but offer additional computational power. However, if for some reason a god thread terminates, that god's team immediately loses.

Initially the team starts with just a god. However, the team can spend treasure to spawn new squids or whales that play on the team.

2.1 The ocean

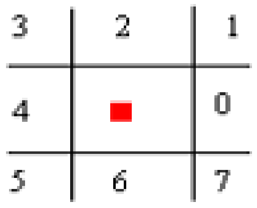
The game is played in a rectangular ocean made of square cells. Each cell can hold exactly one of the following items: a coral, a treasure, a squid, a whale or ink. The ocean is initialized with a map that specifies coral locations. A sample ocean is given in file `sampleocean.sml`.



- Size: We restrict all oceans to be 32x16.
- Cells: Cells on the ocean are identified by 2-dimensional coordinates, with (0,0) being the upper left corner and (31,15) the lower right corner. The first number identifies the left-right (x) coordinate, and the second number identifies the up-down (y) coordinate. Each cell is 1,250 λs by 1,250 λs.

2.2 Directions

Each cell has 8 neighbors. A bot occupying a given cell can move to any of the cell's neighbors. However, a diagonal move takes proportionally longer since it covers more distance. The directions are numbered 0 to 7 as follows:



2.3 Teams

Each team starts a new game with a god and 5 treasure points. Each team is controlled by one RCL program. The team starts with just the god bot, running this program. However, the god can create other bots by spawning.

Different bots have different abilities. Gods can query the game status, spawn new bots, perform computation and share information with their teammates. However, the gods are not physically present in the game and thus cannot move around or interact with the opposing team directly.

Each squid or whale bot occupies a cell in the ocean and faces in some direction. These bots can move around the ocean, fight each other and try to find treasure. Squids can squirt ink that slows down bots that enter it, and whales can destroy squids. Whales must periodically come up to the top of the ocean to breathe.

2.4 Attacks

2.4.1 Squid attack

A squid can squirt ink behind it (i.e., in the opposite direction of which it's facing). The ink temporarily covers 3 cells behind the squid and stays on the cells for `AT_INK_DURATION` clock cycles. After a squid inks, it must go through a cooldown period of `AT_COOLDOWN` clock cycles (to produce more ink) before it can ink again.

If ink lands on a coral, the ink disappears. If ink lands on a squid or whale, the ink disappears, but the bot is immobilized for some time (`AT_SINK_DELAY` for squids, `AT_WINK_DELAY` for whales). Similarly, if a squid or whale swims into ink, the ink disappears, but the bot is immobilized. The breath count of whales immobilized by ink is decremented by `BREATH_CNT_PENALTY`.

2.4.2 Whale attack

If a whale swims into a squid or a squid into a whale, the squid is destroyed.

2.5 Creating new bots

After the game starts, both teams can spawn squids and whales by using the RCL `spawn` expression. Gods are the only bots that can directly spawn more bots. To specify the type of the next bot to be spawned, the god should call the `set type` action using the `do` command.

Newly spawned squids are placed uniformly at random in an empty cell in rows 6 through 10. Newly spawned whales are placed uniformly at random in an empty cell in rows 0 through 5. Red team bots are placed uniformly at random in the leftmost four columns of the ocean; blue team bots are placed uniformly at random in the rightmost four columns.

The input oceans do not have any empty start location completely surrounded by coral. All bots spawn facing north (i.e. in direction 2).

For a team of n bots (counting squids and whales), spawning a squid costs $(n + 3)/3$ treasure points (rounded down) and each additional whale costs $4(n + 3)/3$ treasure points. If a team does not have enough treasure to spawn a bot, the `spawn` expression will fail as described in the RCL specification, and no bot will be created. It is important to note that a bot does not necessarily know if the team has enough treasure left and therefore might try to spawn randomly. It is the responsibility of the world to check that spawn attempts come from gods and that the team has enough treasure to spawn an additional bot of the specified type. Failed spawn attempts do not create a new bot, but they do not otherwise negatively affect the bot that attempts to spawn.

2.6 Collecting treasure

Any bot, except the god, can capture a treasure point for his team by swimming into a square containing that treasure in the ocean. Treasures appear only in empty cells in rows 11 through 15. At the beginning of the game 10 treasures are placed in the ocean uniformly at random. A new treasure appears every `AT_TREASURE` clock cycles, but no more that 15 treasures can be present in the ocean at any given time.

2.7 Scheduling

When each team is allowed to evaluate is important to the fairness of the game. You are going to use your single-step evaluator from Problem Set 5 to evaluate the RCL code for the bots. In each *clock cycle*, every bot of each team is stepped exactly once. The world is then notified that a cycle has ended to that it can perform any necessary updates, such as placing new treasures in the ocean.

2.8 Winning

The game continues for `GAME_LENGTH` clock cycles. Once the time runs out, the team with the largest number of treasure points wins.

3 Implementing actions

The bots have a list of actions that they can take via the `do` command, as described in Figure 1. Each action takes time to execute, and the RCL thread is paused while the action executed. This is accomplished by using the expression (`delay` e `by` n), as described in Part I of the project. The expression returned by a `do` command is a result value wrapped in a `delay` expression that causes the required delay.

Each action returns a value, as described in subsequent sections. The effects of an action are visible to the world and the other bots immediately 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 actions in more detail. Here are some of the constants mentioned in the actions:

`T_EMPTY`	An empty spot
`T_CORAL`	Coral
`T_SQUID`	A squid
`T_OSQUID`	An opponent squid
`T_WHALE`	A whale
`T_OWHALE`	An opponent whale
`T_TREASURE`	A piece of treasure
`T_INK`	Ink

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

move The bot moves from one cell to an adjacent cell in the direction he is currently facing. The bot can move only one cell at a time. Diagonal moves take proportionally larger number of evaluation steps to complete. In general, there are many cases when a move might fail and the bot stays in the same location.

- If a god tries to move, the move fails.
- If any bot tries to move onto a coral, the move fails.
- If any bot tries to move into a treasure, his team's treasure count is incremented, the treasure disappears and the move succeeds.
- If a squid tries to move into a square occupied by another squid, the move fails. If the squid tries to move into a square occupied by a whale, the squid is destroyed.
- If a whale tries to move onto a square that is occupied by a squid, the move succeeds and the squid is destroyed. If the whale tries to move into a square that is occupied by another whale, the move fails.
- If a squid or a whale move into ink, the ink disappears and they become stuck in the ink for the amount of time described before.
- Each time a whale moves, its breath count is decremented (even if the move fails). If the breath count reaches zero, the whale is destroyed. On any step when the whale is at the top row of the ocean, it refills his lungs with air, so its breath count is reset to the maximum value.

turn A bot turns from his current direction to either the left or the right. A turn never fails for a squid or whale. A turn always fails for a god.

ink If the bot trying to ink is not a squid, the action fails. When a squid inks, ink covers 3 squares directly behind the squid.

my status returns the current location, direction and status of the bot. If the bot is a squid, the status is the remaining cooldown clock cycles; and if the bot is a whale, the status is the breath count. If the bot is a god, the action fails.

team status returns the number of treasures, the number of squids and the number of whales for both teams.

scan returns the objects next to the bot in each of the eight directions.

look returns the first item and the distance to that item in some direction from the position of the bot.

inspect returns the object at a particular location.

set type sets the type of the next bot to be spawned (used by gods).

get time queries the number of clock cycles remaining in the game.

Figure 1: Overview of bot actions

Command	Time	Args	Description	Returns	
A_MOVE	AT_SMOVE for squid	None	Move forward	R_SUCCESS	if the move was successful
	AT_WMOVE for whale			R_TREASURE	if a treasure was collected
	AT_GOD for god			R_WON	if a whale destroyed a squid
				R_FAIL	otherwise
A_TURN	AT_TURN	int i	Turn in direction i, where i = T_LEFT or T_RIGHT.	R_SUCCESS	for a squid or whale. If the direction input is invalid, the bot turns left.
	AT_GOD for god			R_FAIL	for a god.
A_INK	AT_INK	None	Squirt ink backwards	R_SUCCESS	if the bot is a squid and his cooldown time period has expired.
				R_FAIL	otherwise.
A_MYSTAT	AT_MYSTAT	None	Returns the status of the squid or whale	$((x, y), d, s)$	where (x, y) is the bot's location, d is the bot's direction. s is the remaining number of clock cycles in the cooldown period for squid or the number of moves remaining until a whale runs out of air.
	AT_GOD for god			R_FAIL	for a god.
A_TEAMSTAT	AT_TEAMSTAT	None	Returns the status of the bot's team and the number of bots and treasures of the opposing team	t	where t is $(c, co, ns, nw, nso, nwo)$. c and co are the number of treasures the bot's and opposing teams have respectively, ns, nw, nso, and nwo are the number of squid and whales on the bot's and the opposing team respectively.
A_SCAN	AT_SCAN	None	Returns the items in the eight cells around the squid/whale.	t	Where t is of the form (n,ne,e,se,s,sw,w,nw). n is the item in the north cell, ne is the item in the northeast cell, etc.
		(int x, int y)	Same action for a god.	t	
				R_FAIL	otherwise.
A_LOOK	AT_LOOK	int i	Returns the nearest object in direction i, where i is 0 through 7 for a squid/whale.	(o, d)	where o gives the type of the first object, and d is the distance to that object.
		((int x, int y),int i)	Same action for a god.	(o, d)	
				R_FAIL	otherwise.
A_INSPECT	AT_INSPECT	(int x, int y)	Returns the type of the object at location (x, y)	o	where o gives the type of the object at the specified location.
A_SET_TYPE	AT_SET_TYPE	int i	Set the type of the next bot to be spawned to whale if i = T_WHALE, otherwise squid	R_SUCCESS	if the bot is a god. If the bot type input was invalid the next bot spawned should be a squid.
				R_FAIL	if the bot is not a god.
A_GET_TIME	AT_GET_TIME		Get the number of clock cycles remaining in the game	t	where t is the number of clock cycles remaining in the game.
A_TALK	AT_TALK	string s	Prints s in the text window	R_SUCCESS	This action has been mostly implemented for you for debugging purposes.

Figure 2: Details of actions

4 RCL extensions

We have provided several RCL features and patterns 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

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

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

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

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

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

4.2 Includes

You may wish to write code that multiple 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 `project` directory and want to include in a bot the `constants.ch` file that we have written, which is in the `cl` directory, you would use

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

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

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

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

4.3 List library

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

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

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

4.4 Booleans

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

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

4.5 Random number generator

A simple random number generator is provided in `rand.ch`:

`seed s`	initializes the generator with seed `s`
`rand n`	Returns a random number between 0 and `n`

4.6 Splay trees

The library `splay.ch` implements a mutable ordered set abstraction, based on splay trees. It has good amortized performance and exploits locality. It uses the lists library, so you need to include `lists.ch` before `splay.ch`.

Each element in the set is identified by a unique key, which may be the same as the element itself. Two elements are considered distinct if they have different keys. Keys are a totally ordered set.

This abstraction can be used to represent an ordinary mutable set if an element is the same as its key. It can represent a mutable map if an element is a pair of its key and a value.

`splay_create order keyOf`	an empty splay tree with key ordering defined by `order` and where the key of element e is `keyOf e`.
`splay_add tree elem alt`	adds the element elem to the tree `tree`. If there is an element already with the same key, it is replaced by the new element and the old element is returned (otherwise `alt`).
`splay_get tree key alt`	is the element in `tree` whose key is equal to `key`, or else `alt` if there is no such element.
`splay_copy tree`	creates a new tree with the same contents as `tree`. This is a fast, constant-time operation that can be used to program in a functional style (if that is desired).
`splay_foldl body init tree`	folds over `tree` applying the function `(body elem curr)` to the elements in ascending order (similar to SML foldl).
`splay_print tree`	prints out the splay tree.

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 in the ocean. This is a string consisting of an event name, followed by a number of parameters, separated by spaces. The following events are recognized by the GUI:

- `"[CONNECTING]"`, initializes the connection with the GUI.
- `"set <image> <x> <y>"`, where `<image>` is a string describing the object image (below is a list of possible images), and `<x>`, `<y>` are two coordinates where the object must be added.
- `"treasures <red treasures> <blue treasures>"`, where `<red treasures>` is an integer representing the number of treasures the red team has and `<blue treasures>` is an integer representing the number of treasures the blue team has.
- `"time <t>"`, where `<t>` is an integer representing the number of clock cycles left in the game.
- `"name <c> <n>"`, where `<c>` is `"red"` or `"blue"` and `<n>` is the name of the team.
- `"chat <c> <message>"`, where `<c>` is `"red"` or `"blue"` and `<msg>` is the message to be printed on the screen.

We require `<x>` and `<y>` to be valid coordinates and `<img>` to be a valid image name. The list of possible image names for `<img>` are listed below, where color is either `"R"` or `"B"` if the bot is on the red team or the blue team respectively. Remember only squids and whales exist in the physical world, so no graphics support for gods!

Image	**Description**
`squidcd`	A squid of color c and facing in direction d
`whalecd`	A whale of color c and facing in direction d
`coral`	A coral
`treasure`	A treasure
`ink`	Ink
`empty`	Empty square

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

```
"set empty 3 7"
"set squidR0 4 7"
"set whaleB4 3 3"
"set coral 19 3"
```

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

[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.

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 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 Project I.

6.2 Designing the world

Your first task is to implement the *Twenty Thousand λs Under the Sea* world in the files `world/world.sml` and `world/game.sml`, and any files you choose to add. Note that you should add files only to the `world` and `cl` directories. You must implement the actions listed in Section 3. You must also make sure that the actions bots take are rendered in the graphic display using the interface detailed in Section 5. You can use the sample bot program we provide to test your world.

6.3 Designing a team

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

6.4 Documentation

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

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

- **Make sure that what is going on in the world matches what is going on in the graphics.** Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember, it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the world is acting properly. It would behoove you to maintain some sort of invariant between the status of the world and the status of the graphics.

- **Problems in the world might actually be problems with the bots.** If you are using your own bots to test the actions and something seems wrong, the bots could just as easily be at fault.

- **Implement and test the actions one at a time.** Don't try to implement all of the actions and test them with one single team. Start with easier actions and work up to the harder ones. For example, start with a simple action like `turn`.

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

6.6 Design review meeting

For this assignment, there will be a *design review meeting* halfway through the assignment. Each group will use CMS to sign up for a meeting, on either April 19, 20, or 22. In the meeting, you are expected to 1) explain the design of your system and give a brief description of the design of your RCL bots and 2) hand in a printed copy of a signatures for each of the modules in your design. Everyone in the group should be prepared to discuss the design and explain why the module signatures are the way they are. We will give you feedback on your design.

In designing module interfaces, think about what functionality needs to go into each module, how the interfaces can be made as simple and narrow as possible, and what information needs to be kept track of by each module.

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

6.7 Final submission

You will submit: 1) a zip file `project.zip` of all files in your `project` directory, including those you did not edit; and 2) your documentation file `doc.txt` (or `doc.pdf`). Although you will submit the entire `project` directory, you should only add new files to the `world` and `cl` folders; the other folder must remain unchanged. If you add new sml or sig files, be sure to modify `sources.cm`.

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

7 Tournament

The CS312 Spring 2007 tournament will be held of May 8, 2007 in Upson B17 from 7:30 to 9:30pm. The tournament is a competition between the RCL bots of the students who wish to compete. Each group may use CMS to submit a RCL team to the tournament. Members of the winning student team will receive a prize and bragging rights. All students are expected to attend the tournament even if they do not submit a team. Members of the course staff may also bring their own teams for post-tournament grudge matches. There will be free food. Not to be missed!

8 Provided source code

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/sampleocean.sml`	Defines a sample ocean
`world/world.sig`	Signature file for handling an action
`world/world.sml`	Functions for handling an action from a bot and implementing all local actions for locking memory
`world/game.sig`	Signature file for the functions contained in the game
`world/game.sml`	Handles the game state
`world/loop.sml`	Starts and continues the main game loop
`net/*.sml`	Network SML code for communicating with the GUI
`gui/*.class`	The GUI class files
`cl/*.ch`	RCL libraries
`cl/random_team.cl`	Sample team program

9 Running the game

You will need Java version 1.5 in order to run the graphics of the game. Although you could run the game without the graphics, it is not recommended, since it is difficult to tell what is going on.

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

1. *Start the GUI.* Start a command prompt (in Windows) or a terminal (in *nix). To start a command prompt in Windows, click on the Start menu, go to Run and type in `command`. At the command prompt, go the the `project/gui` directory and run `"java 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, an ocean will pop up on your screen. The ocean will remain dark until the SML program has connected to it.

2. *Start your SML program.* After you start the GUI, run your SML program (that includes the evaluator, the world, and the networking code that talks to the GUI). Go to the `project` directory and run `CM.make()` to compile the program. Then run `"GameLoop.start(<host-list>, <red team-job>, <blue team-job>)"`, where `<host-list>` is a list of hosts (represented as hostname and port pairs), and `<red team-job>` and `<blue team-job>` are strings representing file names you want to use as red and blue teams. Alternatively, you can run `T.test()`, which starts up a game on `localhost:2005`.

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

10 Written problem: Complexity analysis (10 pts)

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

Elmer Tyes has a idea for how to build a faster mutable ordered set abstraction. Instead of storing all the elements in the sorted array, he will maintain a separate short linked list of up to $m(n)$ elements, where $m(n)$ is some function yet to be determined.

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

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

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

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

c. Elmer wants to make inserts and lookups as cheap as possible. We can actually bring both down to an amortized complexity of $O(\sqrt{n})$. Your goal is to prove this complexity using potential functions. Recall that the amortized complexity of an operation changing the structure s to s' is defined as actual cost of operation + $\Phi(s') - \Phi(s)$.

   Provide a $\Phi(s)$ and a definition of $m(n)$, and use them to show that the complexity of 1 lookup is $O(\sqrt{n})$ and amortized complexity of 1 insert is $O(\sqrt{n})$.

To submit: Turn in a file `complexity.txt` in simple ASCII format containing the solution to this problem.