CS 312 Project (Part II): λ-Craft

Assigned: November 16, 2006

Final submission due: 11:00PM, December 10, 2006
Checkpoint meetings: November 27, 28 and 29, 2006

---

1 Introduction

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

This project places few constraints on how you implement it. This does not mean you can abandon what you have learned about abstraction, style and modularity; rather, this is an opportunity to demonstrate all three in the creation of elegant code. You have to start by laying out a design of how you want to build your system.

For this part there will be a *checkpoint meeting* halfway through the assignment where you will meet with a course staff member to discuss your design. You are required to submit a printed copy of the signatures for each of the modules included in your design.

1.1 Source code

Source code for this project is available in CMS.

1.2 Use of CL

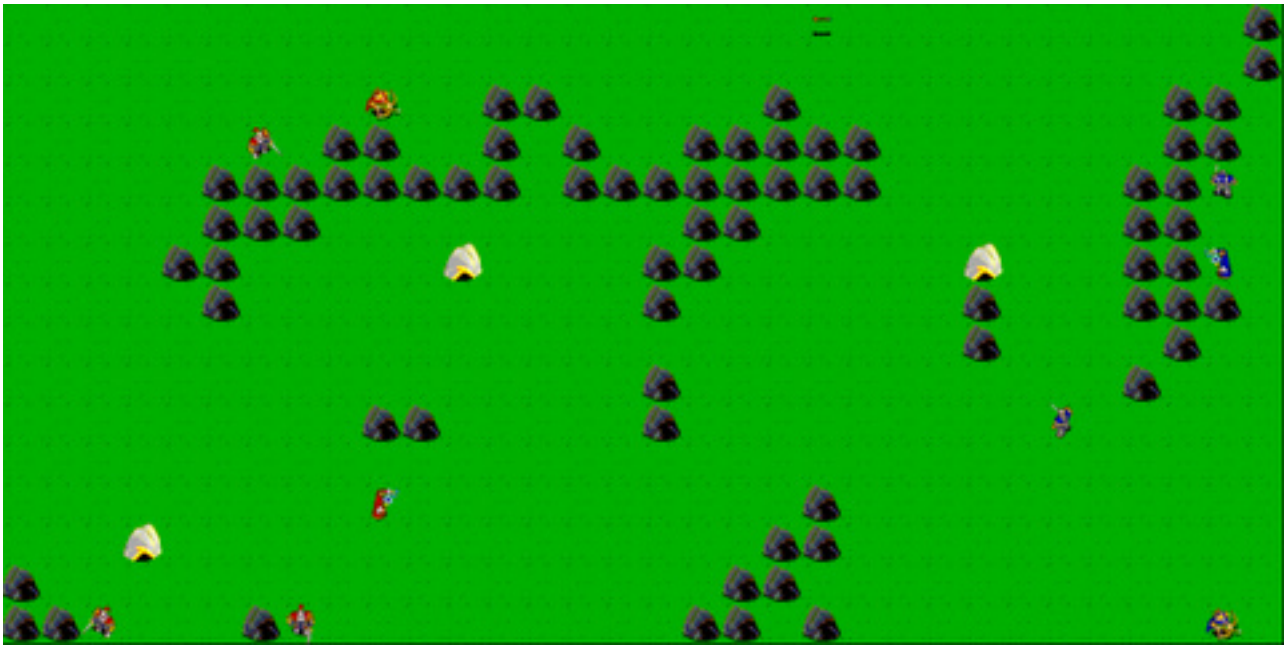
The game of λ-Craft has three kinds of players or units: wizards, archers and knights. (The graphics used for the units are based on graphics from Blizzard Entertainment's WarCraft II.) These units will be driven by a program written in the CL language. You will implement not only the game but also a 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

λ-Craft is a game played by two teams, the red team and a blue team. The object of the game is to capture the enemy wizard. The red wizard is named Zardoz and the blue one - Zodraz. Each team will have archers and knights to help achieve this goal. The first team to capture the enemy wizard wins. If for some reason a wizard program terminates, that wizard's team loses.

2.1 The Field

λ-Craft is played on a rectangular field of square cells. Each cell can hold exactly one of the following items: a mountain, a λ-mine, a wizard, an archer, a knight or an archer's arrow. The field is initialized with a map that specifies mountain locations. An example field is given in file `samplefield.sml`.



- Size: We will restrict all fields to be 32x16.
- Cells: Cells on the field are identified by 2-dimensional coordinates, with (0,0) being in the upper left corner and (31,15) in the lower right corner. The first number identifies the left-right (x) coordinate, and the second number identifies the up-down (y) coordinate.
- Mountains: mountain are placed at certain location on the field. They are stationary and no character can walk through or occupy the same cell as a mountain.
- Starting point: Zardoz and Zodraz will start at opposite sides of the field facing each other. Zardoz will be placed at (0,8) and Zodraz will be at (31,7). The input map will not have mountains on the start locations of the wizards nor will any start location be completely surrounded by mountains.

2.2 Directions

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

3	2	1
4	■	0
5	6	7

7	6	5
0	■	4
1	2	3

2.3 Teams

Each team starts a new game with a wizard and enough λs to spawn exactly 5 units. Each team is controlled by one CL program. The first unit spawned is the team's wizard. Each unit occupies a location on the field and has a direction in which it is facing. The units can move around the field, battle each other and try to capture the enemy wizard.

2.4 Attacks

2.4.1 Wizard attack

Zardoz and Zodraz are all-powerful wizards who can temporarily freeze any unit (even a teammate) at any location on the field. If he aims at a location with a knight or an archer, the unit will freeze for 1.5 $\times$ number of cycle it takes to move. If he aims at a location with an arrow, then the arrow is destroyed. This attack drains some of the wizards mana, so the wizard cannot perform any other action or computation for the number of cycles it takes to move. Both wizards have a force field that protects them against any freeze attacks.

2.4.2 Archer's attack

An archer can shoot arrows by invoking the shoot action. The arrow travels at a speed four times that of the units. It can travel a distance of up to eight squares. If the arrow hits a knight or an archer, the unit is destroyed with probability (0.21 - 0.01x), where x is the number of squares the arrow has traveled. If the arrow hits a mountain, a λ-mine or a wizard then it is destroyed. If the arrow collides with another arrow, then both arrows are destroyed.

2.4.3 Knight's attack

A knight attacks if he tries to move onto a square that contains an enemy knight or archer. If the knight's opponent has his back to the knight (i.e. facing in the same direction as the knight), he is destroyed with probability 0.7. If the opponent is facing sideways, he's destroyed with probability 0.5. If he's facing towards the knight (i.e. in the opposite direction as the knight), he is destroyed with probability 0.35.

2.5 Acquiring new units

After the game starts, both teams can spawn more archer or knight units by using the `spawn` command in CL. The wizards are the only units that can directly spawn more units. To specify the type of the next unit to be spawn, the wizard should call the set type action using the `do` command. A unit spawned appears in a nearest free tile to his team's wizard. If an team has not lifted enough λs, then its wizard cannot spawn new units. It is important to note that a unit does not necessarily know if the team has enough λs and therefore might try to spawn randomly. It is the responsibility of the world to check if a unit is a wizard and if that wizard's team has enough λs to spawn an additional unit. If this check fails, then the unit who attempted the spawn should proceed normally. If a wizard is in an team of $x - 1$ units, then the x^{th} unit costs x λs.

2.6 λ lifting

Any unit can lift λs from a λ-mine that is located on the field. At the beginning of the game three λ-mines are placed randomly on the field. Each mine contains six λs. After the mine is emptied, it disappears and a new λ-mine appears immediately at a different, randomly generated location such that there is one square between the location of the mine and any unit.

2.7 Scheduling

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

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

2.8 Actions

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

Here is a description of the possible actions.

move The unit moves from one cell to an adjacent cell in the direction it is currently facing. The unit can move only one cell at a time. In general, there are many cases when a move might fail and the unit will stay in the same location.

- None of the units are trained to climb mountains. So if any unit tries to move onto a mountain, the move fails
- If any unit tries to move onto a λ-mine, he lifts a λ from the mine, but remains in his original location.
- If an archer or a knight moves onto an arrow, then the arrow hits the unit and he is destroyed with the probability described above. If a wizard tries to move onto an arrow the arrow is destroyed. In both cases if the unit survives, then the move succeeds.
- If an archer tries to move onto a square that is occupied by another archer or knight then the move fails.
- If a knight tries to move onto a square that is occupied by an archer or knight of the opposing team, this is considered an attack. (If the unit is on the attacker's team, the move fails.) If the knight destroys his opponent, he is moved onto his opponent's square.
- If a wizard tries to move onto a square occupied by another unit, the move fails.
- If a knight or archer moves onto a square with an enemy wizard, the unit captures the wizard and the game ends. If the wizard is on the unit's team, then the move fails.

turn A unit turns from his current direction to either the left or the right. A turn never fails; if the direction passed in is invalid, the unit turns in right or left arbitrarily.

shoot If the unit trying to shoot is not an archer, the shoot fails. When an archer shoots, the arrow leaves the archer in the direction the archer is currently facing. If the next square in the direction an archer is facing is occupied, then the result is as described in Archer's attack section.

freeze The wizard can freeze a certain location on the field. If the unit trying to freeze is not a wizard, the freeze fails. If the location is occupied by a unit that's not the enemy wizard, the freeze succeeds. If the location has an arrow, the arrow is destroyed. Otherwise, the freeze fails.

my status returns the current location and direction of the unit.

team status returns the number of λs the unit's team has, the number of units on his team, the location of his wizard, the number of units on the enemy team, and the list of the PIDs (process IDs) of all units currently alive on his team.

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

look returns the first object and the distance to that object in some direction from the position of the unit.

inspect returns the object at a particular location.

set type sets the type of the next unit to be spawned.

3 Implementing actions

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

Each action returns an *Action-Specific Return* (ASR), as described in subsequent sections. The results of an action should be visible to the world and the other units when the action is performed and before the unit is made to wait.

3.1 Possible actions

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

T_EMPTY	Empty spot
T_MOUNTAIN	A mountain
T_WIZARD	The one (of two), the (almost) only, the most powerful Zardoz! (or Zodraz)
T_OWIZARD	The enemy wizard
T_ARCHER	An archer
T_OARCHER	An enemy archer
T_KNIGHT	A knight
T_OKNIGHT	An enemy knight
T_LAMBDAMINE	A λ-mine
T_ARROW	An arrow

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

Command	Base Time	Args	Description	ASR	
A_MOVE	AT_MOVE	None	Move forward	R_SUCCESS	if the move was successful
				R_LIFTED	if the unit tried to move into a λ-mine
				R_WON	if a knight tried to move onto an enemy archer or knight and won the fight, then the knight is moved onto the enemy unit's square
				R_LOST	if a knight tried to move onto an enemy archer or knight and lost the fight, then the knight is not moved
				R_FAIL	otherwise
A_TURN	AT_TURN	Int i	Turn in direction i, where i = T_LEFT or T_RIGHT.	R_SUCCESS	in all cases. If the direction passed in was invalid the unit will turn left.
A_SHOOT	AT_SHOOT	None	Shoot an arrow forward	R_SUCCESS	if the unit is an archer
				R_FAIL	if the unit is not an archer
A_FREEZE	AT_WIZ_FREEZE	(x,y)	Freeze location (x,y)	R_SUCCESS	if the unit is a wizard and location (x,y) contains an archer or knight
				R_DESTROYED	if the unit is a wizard and location (x,y) contains an arrow
				R_FAIL	if the unit is not a wizard or the location (x,y) contains another item
A_MYSTAT	AT_MYSTAT	None	Returns the status of the unit	$[(x, y), d]$	where (x, y) is the unit's location, d is the direction the unit is currently facing
A_TEAMSTAT	AT_TEAMSTAT	None	Returns the status of the unit's team and the number of units on the enemy team	$list$	where $list$ is $[l, n, no, pid_w, (x, y), pids_a, pids_k]$ and l is the number of λs the team has, n and no is the number of units on the player's and the enemy's team respectively, pid_w and (x, y) are the wizard's pid and position, $pids_a$ is a list of archer pids, $pids_k$ is a list of knight pids
A_SCAN	AT_SCAN	None	Returns the items in the eight cells around the unit.	l	Where l is of the form [n,ne,e,se,s,sw,w,nw]. n is the object in the north cell, ne is the object in the northeast cell, etc.
A_LOOK	AT_LOOK	Int i	Returns the nearest object in direction i, where i = DIR_N, DIR_NE, DIR_E, etc.	[o, d]	where o gives the type of the first object, and d is the distance to that object.
A_INSPECT	AT_INSPECT	(Int x, Int y)	Returns the type of the object at location (x, y)	o	where o gives the type of the object at the specified location.
A_TALK	AT_TALK	String s	Prints s in the text window	R_SUCCESS	
A_SET_TYPE	AT_SET_TYPE	Int i	Set the type of the next unit to be spawned to knight if i = T_KNIGHT, otherwise archer	R_SUCCESS	if the unit is a wizard. If the unit type input was invalid the next unit spawned should be an archer.
				R_FAIL	if the unit is not a wizard

Figure 1: List of Actions

4 CL Extensions

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

4.1 Recursive functions

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

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

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

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

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

4.2 Includes

You may wish to write code that multiple CL programs can use. You can now do this using the `#include` command. The argument is the name of the file to include. Keep in mind that the path for the file should be relative to the directory *from which you run SML*, not the directory in which the CL file is located. If you execute SML from the project directory and want to include in a unit the constants.ch file that we have written, which is in the cl directory, you would use

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

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

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

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

4.3 List library

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

functions:

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

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

4.4 Other libraries

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

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

Finally, 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`

5 Java GUI

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

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

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

- `"[CONNECTING]"`, initializes the connection with the GUI.
- `"set <image> <x> <y>"`, where `<image>` is a string describing the object image (below is a list of possible images), and `<x>`, `<y>` are two coordinates where the object must be added.
- `"lambdas <red lambdas> <blue lambdas>"`, where `<red lambdas>` is an integer representing the number of λs the red team has and `<blue lambdas>` is an integer representing the number of λs the blue team has.

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"`, `"B"` or `"F"` if the unit is on the red team, the blue team or frozen respectively. Remember only archers and knight may be frozen so there is no graphics support for frozen wizards!

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

Image	Description
wizardcd	A wizard of color c and d is the integer direction the wizard is facing
archercd	An archer of color c and d is the integer direction the archer is facing
knightcd	A knight of color c and d is the integer direction the knight is facing
mountain	A mountain
lambdamine	A λ-mine
arrow	An arrow

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

```
"set empty 3 7"
"set wizardR0 4 7"
"set archerB4 3 3"
"set knightF4 19 3"
```

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

6 Your tasks

There are several parts to the implementation of this project. Make sure you spend time thinking about each part before starting. Start on this project *early*. There are many things you will have to take into consideration when designing the code for each section.

6.1 CL interpreter

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

6.2 Designing the world

Your first task is to implement the λ-Craft world in the files `world/action.sml` and `world/game.sml`, and any files you choose to add. Note that you should add files only to the `world` and `cl` directories. You must implement the actions listed in Section 3. You must also make sure that the actions units take are rendered in the graphic display using the interface detailed in Section 5. You can use the sample unit program we provide to test your world.

6.3 Designing an team

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

6.4 Documentation

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

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

6.5 Things to keep in mind

Here are some issues to keep in mind when designing and implementing the world:

- **Think carefully about how to break up your program into loosely coupled modules.** The program will be complex and difficult to debug unless you can develop modules that encapsulate important aspects of the game. Design the interfaces to these modules carefully so that you can work effectively with your partner and can do unit testing of the modules as you implement.

- **Make sure what is going on in the world and what is going on in the graphics match.** Updating one does not automatically update the other. If you are watching the game and something seems to go wrong, remember, it could just be the code controlling the output to the screen. Moreover, just because the graphics look correct doesn't mean the world is acting properly. It would behoove you to maintain some sort of invariant between the status of the world and the status of the graphics.
- **Problems in the world may actually be problems with the units.** If you are using your own units to test the actions and something seems wrong, the units could just as easily be at fault.
- **It is best to implement and test the actions one at a time.** Don't try to implement all of the actions and test them with one single team. Start with the easier actions and work up to the harder ones. An actions like `turn` is probably easier to implement relative to the other actions.

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

6.6 Checkpoint meeting

For this assignment, there will be a *checkpoint meeting* halfway through the assignment. These meetings will be held on November 27, 28, and 29. You are expected to 1) explain the design of your system and give a brief description of the design of your CL units and 2) hand in a printed copy of a signatures for each of the modules in your design.

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

6.7 Final submission

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

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

7 Tournament

Sometime during finals week, most likely on December 12, 2006, we will hold a competition between the CL players of the students who wish to compete. Each group may submit a CL team that will play against other students teams. Details on the tournament time and location and the submission procedure will be available later.

8 Given Files

Many files are provided for this assignment. Most of them, you will not need to edit at all. In fact, you should only edit and/or create new files in the `/world` and `/cl` directories. Here is a list of all the files and their functions.

`gfx/*`	Graphics files for the GUI
`world/samplefield.sml`	Defines a sample field
`world/action.sig`	Signature file for handling an action
`world/action.sml`	Functions for handling an action from a unit and implementing all local actions for locking memory
`world/game.sml`	Handles the game state
`world/game.sig`	Signature file for the functions contained in the game
`world/loop.sml`	Starts and continues the main game loop
`net/*.sml`	Network SML code for communicating with the GUI
`gui/*.class`	The GUI class files
`cl/*.ch`	Libraries for lists and booleans
`cl/sampleunit.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 would be nearly impossible to tell what is going on.

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

1. *Start the GUI.* Start a command prompt (in Windows) or a terminal (in *nix). To start a command prompt in Windows, click on the Start menu, go to Run and type in `command`. At the command prompt, go the the `project/gui` directory and run `"java Gui <host> <port>"`. This tells the graphics program to start up and connect to the SML world running on `<host>` at port `<port>`. The `<host>` and `<port>` parts are optional. The default (if you run `"java Gui"`) is `localhost:2005`. Once the GUI is started, a field will pop up on your screen. The field will remain dark until the SML program has connected to it.

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

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