CS 312 Project (Part II): ML-Man

Assigned: November 21, 2005

Final submission due: 9:00AM, December 14, 2005
Checkpoint submission: 11:59PM, November 30, 2005

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 ML-Man, based on the arcade game Pac-Man. In terms of programming, you will have to implement the world for this game (in SML), as well as the code for the characters (in CL). We have provided some graphical support that you can use to display the progress of the game graphically. You will keep the same partner you had for part I; consult Prof. Pingali if this is exceptionally problematic.

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

For this part there will be a *checkpoint submission* halfway through the assignment. You are required to submit a document that describes the design of your system and as much of the remaining code as you have by that point.

1.1 Source code

Source code for this project is available in CMS.

1.2 Use of CL

The game of ML-Man has two kind of characters or players: MLman and the ghosts. These players will be driven by programs written in the CL language. You will implement not only the game but also at least one MLman and one ghost program 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

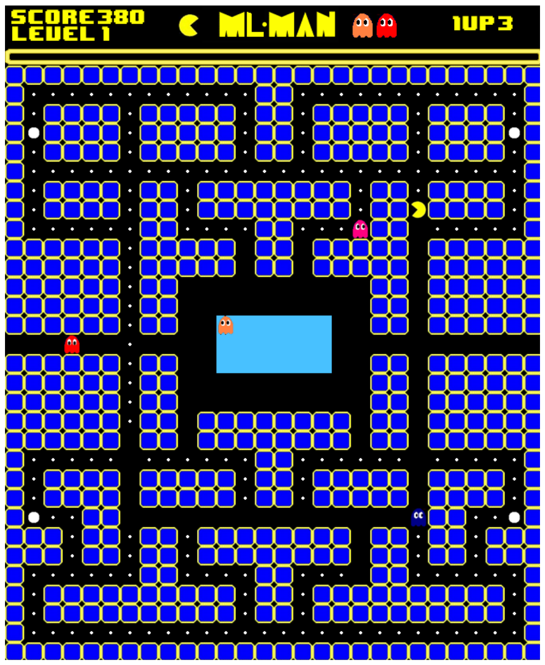
2.1 Players

The game consists of two kind of players:

- MLman: Its goal is to collect as much points as possible. It can collect points by eating dots, power pills, and ghosts whenever it can.
- The ghosts: Their goal is to intercept and capture MLman. There are four ghosts in the game: Zardoz(red), Eager (blue), Greedy (pink), Curried (orange)

2.2 The Board

An example of a ML-Man board is depicted below:



Any maze contains halls divided by walls. MLman and the ghosts navigate around the the halls, which may contain dots or power pills in the floor. The maze may contain tunnels. Tunnels are shortcuts that let MLman wrap around the board. A tunnel entry is an empty space, the exit is another empty space at another point in the maze. In the example above there is one tunnel in the middle row. A special area called the `pen` is a box where all ghosts start. Ghosts can exit the pen at appropriate times, and they come back to it when they are captured by MLman. Neither MLman nor a live ghost can enter back to the pen. MLman starts directly near the middle of the board, the accurate position is a parameter in our game.

2.2.1 Specifiying the board and starting points

The board shown above is specified in file `sampleboard.sml`.

- Size: We will restrict all mazes to have size 28x31.
- Cells: Cells on the board are identified by 2-dimensional coordinates, with (0,0) being in the upper left, and (27,30) in the lower right corner. The first number identifies the left-right (x) coordinate, and the second number identifies the up-down (y) coordinate.
- The pen: The pen is defined by a rectangle ((xt,yt),(xb,yb)) and a door (xd,yd) which is part of the border of the rectangle. The cell (xt,yt) is the top left corner, (xb,yb) is the bottom right corner of the pen, and (xd,yd) is the door where ghosts can come out from the pen. All cells in the border of the pen (except the door) must be a wall. In the example above, the pen is defined by the rectangle ((11,13), (16,15)), and door (14,13).
- Tunnels: Tunnels are specified by directed pairs ((xin,yin), (xout,yout)), where (xin,yin) is the entry of the tunnel, and (xout,yout) is the exit. Since tunnels must be in the border, either xin, yin must be equal to a border (0 or boardsize - 1). Similarly with xout and yout. In the example above, the horizontal tunnel is specified by two pairs: one specifying the tunnel from left to right ((0,14),(27,14)), the other from right to left ((27,14),(0,14)).
- Dots, pills, walls: The rest of the maze is specified by a list of coordinates where there are dots on the floor, power pills and walls. All these sets of coordinates must be disjoint.
- Halls: Any other cell not specified in any of the above is considered an empty space.
- Starting point: In the example above, MLman starts in position (14,23), ghosts always start inside the pen (position (14,15) in this example).

2.3 Playing the game

The game starts with one single CL program for MLman, and a single CL program for the ghosts.

After the game start, the ghost program can spawn more ghosts using the `spawn` command in CL. There is a limit of 4 ghosts allowed at a time. The game engine is responsible for checking this limit, if a ghost tries to spawn a new ghost, and there are already 4 ghosts in the game, the game engine will terminate the spawn process, and allow the parent process to continue running.

All ghosts start in the pen. To give MLman some advantage, The ghosts wait until ML-man makes a certain number k_i of moves (see definitions.sml). The first ghost waits is always free, the second waits k, the third waits $2k$, and the last waits $3k$. MLman starts with 3 lives, and 0 points.

The goal for MLman is to collect the maximum number of points within his number of lives. MLman can move around the board and eat dots (10 points each). Whenever he eats a power pill (50 points) he becomes energized for 600000 cycles. During that time, he can chase ghosts and capture them too. The ghosts are worth 700 points.

The goal of the ghosts is to capture MLman as fast as possible, preventing him from collecting points. The ghosts can wander around and try to find MLman. If MLman is not energized they can capture MLman (making MLman lose a life, and restarting every player to their original position). When MLman is energized, they try to run away (to avoid being captured). They run two times slower when MLman is energized. If they are captured they become harmless, and they need to find their way back to the pen in order to become scary ghosts again.

When MLman is captured and the players are restarted, the board of the game is not changed. When MLman finishes eating all dots and power pills, the level is completed and a new level is loaded. All levels have the same configuration, same starting point for all players, same number and location for dots and power pills.

2.4 Actions

MLman and the ghosts interact with the world by using the `do` command. Each action takes a certain number of clock cycles. A *clock cycle* is a single evaluate step for all of the characters. In other words, if a ghosts performs an action that takes 100 clock cycles, it takes the time of that specific ghost performing 100 evaluation steps, not $\frac{100}{\text{No. of players}}$ steps. The amount of time each player is allowed to evaluate is important to the fairness of the game. You are going to use your single-step evaluator from Project Part I to evaluate the CL code for the players. The order of the evaluation must be fair so that each ghost and MLman are treated equally. In every clock cycle, every player is stepped exactly once; the world is then notified that a cycle has ended, so that it may update information.

Here is a description of the possible actions.

move	The player moves from one tile to an adjacent tile in the direction it is currently facing. The player can only move one tile at a time. If any player attempts to move into a wall, the move fails and the player stays in its current location. Same happens when a ghost tries to enter a tunnel, or when players try to enter/exit the pen and they are not allowed. When a ghost and MLman meet in the same tile, either MLman is captured (if he was not "energized") or the ghost is captured (if MLman was "energized").
turn	A player turns from the current direction to either the left, the right, or 180^o. A turn never fails; if the direction passed in is invalid, the player turns left.
my status	returns the current status for a player. This includes the coordinates of the player, the direction the player is currently facing, whether the MLman is currently energized, and (if it is a ghost) whether it has been captured.

scan	returns the objects next to the player in the four cardinal directions. This can be used for a player to detect corners and intersections of halls, entrance to a tunnel, door of the pen.
look	returns the first object (and the distance to that object) in some direction from the position of the player.
look no dots	returns the first wall, ghost or MLman (and the distance to it) in some direction from the position of the player.
inspect	returns the object at a particular location.
nearest dots	(only for MLman) returns the location of the five nearest dots.
find power pills	(both MLman and ghosts) returns the location of the remaining power pills.
find ghosts	(only for MLman) returns the location of the ghosts.
find pen	(only for ghosts) returns the coordinates of the door to the pen.

3 Implementing actions

The players 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 player that calls an action gets returned to it the return value, wrapped in a `delay` expression which produces the required delay.

Each action returns an *Action-Specific Return* (ASR), as described in subsequent sections.

The results of an action should be visible to the world and the other players when the action is performed and before the player 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_WALL	A wall
T_TUNNEL	A tunnel entrance
T_GHOST	A ghost
T_GHOST_V	A vulnerable ghost
T_MLMAN	The only, the greatest, the man, MLman!
T_DOT	A edible dot piece
T_POWERP	A chewy and delicious power pill

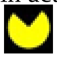
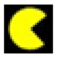
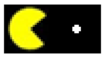
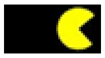
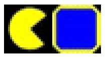
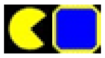
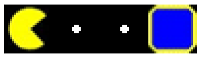
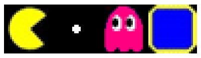
These and other constants are defined in cl/constants.ch.

Command	Base Time	Args	Description	ASR	
A_MOVE	AT_MOVE	None	Move forward	R_SUCCESS R_HITWALL	if the move was successful if the player tried to move into a place where it cannot (e.g. in front of a wall, or a ghost trying to get out of the pen too early)
A_TURN	AT_TURN	Int i	Turn in direction i, where i = T_LEFT, T_RIGHT, or T_BACK.	R_SUCCESS	in all cases. If the direction passed in was invalid (neither of the specified values) the player will turn left.
A_MYSTAT	AT_MYSTAT	None	Returns the status of the player	$[(x, y), d, s]$	where x, y are the player's coordinates, d is the direction the player is currently facing, s an integer indicating some state. For MLman, 1=energized, 0=not energized. For ghosts, 0=normal, 1=vulnerable,2=harmless.
A_SCAN	AT_SCAN	None	Returns the objects (including dots and power pills) in the four cells around the player.	[n,e,s,w]	Where n is the object in the north cell, s in the south cell, w west cell and e east cell of the player. Each object can be T_WALL, T_EMPTY, T_TUNNEL, T_GHOST, T_MLMAN, T_DOT or T_POWERPILL.
A_LOOK	AT_LOOK	Int i	Returns the nearest object in direction i, where i = T_LEFT, T_RIGHT, T_BACK, T_FORWARD	[o, d]	where o gives the type of the first item, and d is the distance to that item.
A_LOOKNODOT	AT_LOOKNODOT	Int i	Returns the nearest object (excluding dots and power pills) in direction i	[o,d]	like T_LOOK but ignoring dots and power pills.
A_INSPECT	AT_INSPECT	(Int x, Int y)	Returns the type of the object at location (x, y)	T_POWERP, etc.	where the return value gives the type of the object at the specified location.
A_NEAR_D	AT_NEAR_D	None	Find the closest five dots using Cartesian distance as the metric.	l R_FAIL	l is a list of coordinates of the five nearest dots (the list can be smaller if there are less than 5 dots in the board) if current player is a ghost.
A_FINDPO	AT_FIND_PO	None	Find all power pills.	l	l list of coordinates for each power pill in the board. l is empty if there is none.
A_FINDGH	AT_FIND_G	None	Find all ghosts in the board.	l R_FAIL	l list of coordinates for each ghost in the board and outside the pen. l is empty if there is none. if current player is a ghost.
A_FINDPE	AT_FIND_PE	None	Find pen door. Returns its coordinates.	(x,y) R_FAIL	coordinates of the pen's door if current player is MLman.

Figure 1: List of Actions

3.2 Examples of actions

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

	`do (A_TURN,T_RIGHT)`	⇒		`R_SUCCESS`
	`do A_MOVE`	⇒		`R_SUCCESS`
	`do A_MOVE`	⇒		`R_HITWALL`
	`do (A_LOOKNODOT, T_FORWARD)`	⇒		`(T_WALL, 3)`
	`do (A_LOOKNODOT, T_FORWARD)`	⇒		`(T_GHOST, 2)`

4 CL Extensions

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

4.1 Recursive functions

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

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

This function can be implemented in CL as follows:

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

4.2 Includes

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

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

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

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

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

4.4 Boolean library

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

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

5 Java GUI

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

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

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

- `"[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.

[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.

- `"status <level> <lives> <score> <energizedsec>"`, where `<score>` is an integer representing the current score of MLman, `<level>` is the number of boards completed by the MLman player, `<lives>` is the number of lives remaining, and `<energizedsec>` is the number of seconds that MLman has left being energized . (GUI limitations: to display correctly, lives must be less than 99, score less than 99999, and energized time is between 0-60).

Note: You should send as few messages as possible, otherwise the GUI will be extremely slow. For example, only send the status message when the score changes, or when the energized time changes (every 10000 steps), and not on every step.

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. For ghosts, one can specify a modifier *c* to denote a color/state (b: blue, r: red, p: pink, o: orange, w:white (vulnerable), x:eyes only (harmless)). For any player, *i* is one directions (u: up, d: down, l: left, r: right):

Image	**Description**
ghost*ci*[p]	A ghost of color *c* looking in direction *i* (p is optionally used for ghosts inside the pen)
mlman*i*	MLman looking in direction *i*
mlmanDead	MLman when is captured
dot	A dot piece
power	A power pill
wall	A wall
empty	An empty space
pen	An empty space inside the pen

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

```
"set empty 3 7"
"set mlmanu 4 7"
"set empty 3 2"
"set ghostbl 3 3"
```

When a player moves, you have to restore the information in the old position (e.g. when a ghost moves over a dot, the dot must stay there after). Then, set the new position for the player. Is important that you restore the old positions all at once, and then set the new positions. Otherwise, restoring the old position of a player may erase the image of another player 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 PS5.

6.2 Designing the world (70pt)

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

6.3 Designing a player (20pt)

Design a MLman in the file `cl/mlman.cl`, and a group of ghosts in file `cl/ghosts.cl`. Your players should be able to perform reasonably good. If your MLman can beat the sampleghosts included in the release, and your ghosts can capture our samplemlman, you should be able to get most of the credit for this portion of the project.

6.4 Documentation (10pt)

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 any changes to or 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 MLman and ghosts, 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 players.** If you are using your own players to test the actions and something seems wrong, the players 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 player. 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 ghost team. Consider, for instance, that your ghosts can communicate. Your ghosts share a global memory that MLman cannot access. Use it to your advantage to coordinate your ghosts movements.

6.6 Checkpoint submission

For this assignment, there will be a *checkpoint submission* halfway through the assignment. You are expected to submit a 1-2 page document `design.txt` (or `design.pdf`) that explains the design of your system. You should decide all additional files you will need, and the signature for each of these new files. You will submit your design by November 30, at 11:59pm.

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

6.7 Final submission

You will submit: 1) a zip file `world.zip` of all files in your `world` directory, including those you did not edit; 2) a zip file `cl.zip` of all files in your `cl` directory; 3) your documentation file `doc.txt` (or `doc.pdf`); and 4) the file `sources.cm` in case you have additional sig and sml files to include.

We should be able to unzip your code, copy your sources.cm file and run `CM.make()`, and be able to use your implementation with no compilation errors.

7 Tournament

At the end of the semester, there will be a competition between the programs of students who wish to compete. Each student project group may submit a MLman and a ghosts program that will play against other student projects. Details on the tournament time and location and the submission procedure will be available later.

8 Given Files

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

`gfx/*`	Graphics files for the GUI
`world/sampleboard.sig`	Signature file for specifying a board
`world/sampleboard.sml`	Defines a sample board
`world/action.sig`	Signature file for handling an action
`world/action.sml`	Functions for handling an action from a player and implementing all local actions for locking memory
`world/game.sml`	Handles the game state
`world/game.sig`	Signature file for the functions contained in the game
`world/loop.sml`	Starts and continues the main game loop
`net/*.sml`	Network SML code for communicating with the GUI
`gui/*.java`	The java files that implement the GUI
`gui/*.class`	The GUI class files
`cl/*.ch`	Libraries for lists and booleans
`cl/sampleMlman.cl`	Sample MLman
`cl/sampleGhosts.cl`	Sample ghosts

9 Running the game

You will need Java version 1.4.2 or later 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 board will pop up on your screen. The board will remain dark until the SML program has connected to it.

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

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