**Structure and Interpretation of Computer Programs**

**15CSE402**

**Maze Game**

**By:**

**Rajendra Prasath S S**

**Eswar V**

**Vishal Varma K**

**Chapter 1** **Introduction**

**Problem Statement**

The task is to identify few concepts read as part of the SICP and design a graphical way of explaining those concepts in the spirit of the tutorial Quick: An Introduction to Racket with Pictures*,* implement them and provide descriptive explanation.

1. Identification of at the least 3-4 concepts from SICP course to be demonstrated graphically.
2. Expected contribution
   1. The concept and its explanation
   2. The graphical implementation of the concept
   3. The output with suitable explanation emphasizing the concept

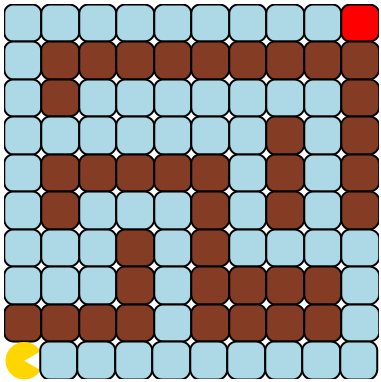
**Problem Definition**

The maze game is a game of guiding the protagonist from the source to the destination crossing series of obstacles like walls and bombs. The obstacles are defined as follows:

* Walls – obstructs movement of the protagonist in that direction
* Bombs – secret mines that kills the protagonist

The protagonist can move in the four directions: up, down, left, and right. The aim of the player is to guide our protagonist across several of the obstacles to reach the destination. It can be noted that there may be several paths that can reach to the destination. As the protagonist reaches the destination successfully, the player is provided with the score (number of steps to reach the destination).

This game is a graphical representation of classical maze games that employs several SICP concepts at various levels to accomplish certain behaviors.

The following characters are graphically represented in the maze to the left:  
(a) Protagonist – Yellow Pacman  
(b) Brown Cells – Walls  
(c) Blue Cells – Paths  
(d) Red Cell – Destination  
  
It should be noted that bombs are invisible to the players and hence players must be careful while moving across the maze.

**Chapter 2** **Prerequisites and Basic Visualizations**

**Prerequisite Libraries and Packages**

Use of #lang slideshow and (require 2htdp/image):

* Supports use of basic image construction functions and allows complex image structures built using the basic construction functions. The basic image functions include circle, ellipse, polygons, texts, and bitmaps.
* Allows image manipulation functions like rotations, overlays and flipping.
* The package and tools are extensively used to visualize and manipulate images in various degrees.

**Basic Visualizations**

1. **The protagonist – Pacman**





The parameters required to create Pacman are explained as follows:

* Wedge
  + First parameter – radius of the wedge
  + Second parameter – angle of the wedge
  + Third parameter – solid or outlined image
  + Fourth parameter – color of the wedge
* The wedge is rotated 30 degrees to make it look like Pacman

1. **The paths**





The parameters required to create path are explained as follows:

* Filled rounded rectangle
  + First and second parameter – width and length
  + Third parameter – color
  + Fourth parameter – border color
  + Fifth parameter – border width

1. **The walls**





The parameters required to create Pacman are explained as follows:

* Filled rounded rectangle
  + First and second parameter – width and length
  + Third parameter – color
  + Fourth parameter – border color
  + Fifth parameter – border width

**Chapter 3** **Conceptual Implementations**

**Higher Order Procedures:**

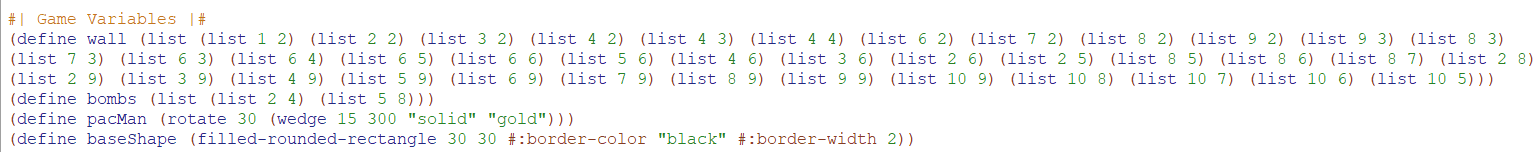
Higher Order Procedures (HOP) are procedures that accepts other procedures as parameters and/or returns a procedure as output.



Above code is implementation of basic walls, paths and bombs. Here, sqr is a procedure that defines a filled square with rounded borders. A procedure (lambda procedure) is returned as output and hence it is a higher order procedure.

**Local Binding:**

Local binding allows to define variables that are only visible to well defined sections of the program rather than being visible globally. This is achieved in racket using either define, let, or let\*.



The game variables i.e., pacman, bombs, walls, paths are all required throughout the execution of the entire program. Thus, they are defined globally so that they can be accessed any time.



This x is used to construct the entire maze and is only needed to build the maze. Thus, let is used to define x only within the scope of row procedure that is used to build the maze.

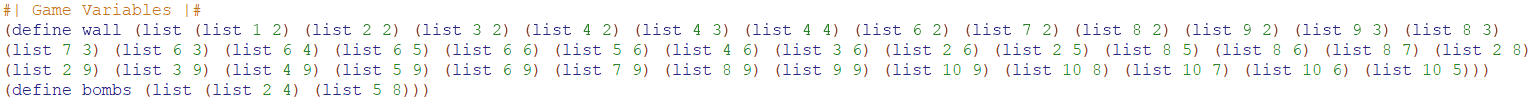


These variables are used to move the pacman across the maze. They are computed with the help of player moves (up, down, left, or right) and thus are required only to update pacman position and rebuild the maze.

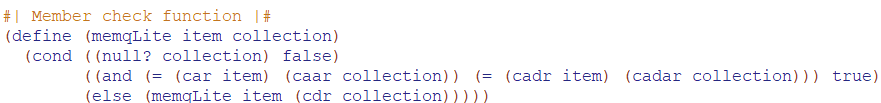
It must be noted that let\* allows variables defined within it to be used to define other such variables and this feature differentiates let\* from let.

**Data Structures:**

Data structures are complex data, defined by combination of primitive data types. Here, to maintain details of coordinates of walls and bombs and list of lists is maintained as variables.



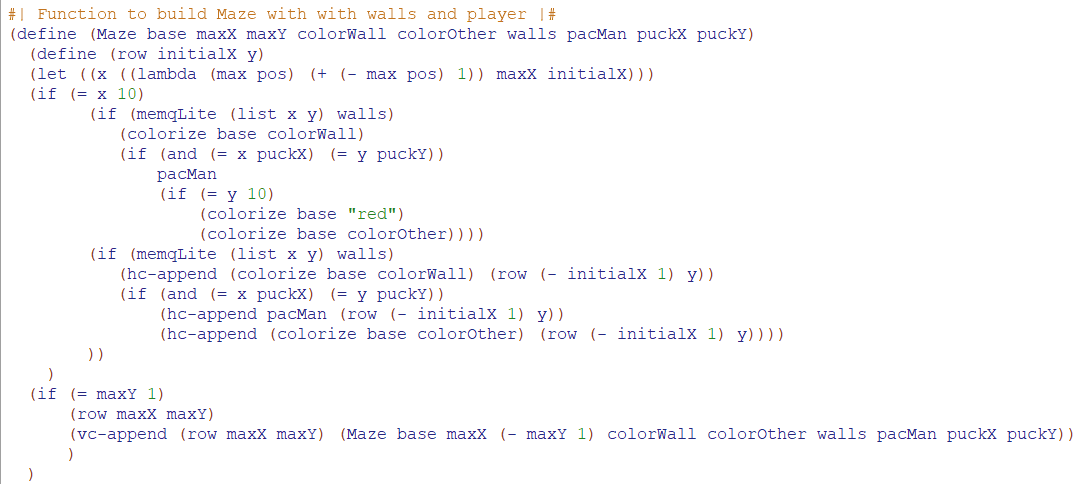
These coordinates are relative w.r.t source of the maze (starting location of the pacman). There are also supporting functions to check if a given coordinate pair is wall or a bomb. This is achieved using a modified version of the memq procedure.



This procedure checks if given coordinate (a list item) is present in the list of lists (collection of wall coordinates). To access x and y coordinates of a particular cell in the maze, shorthand notations of car and cdr are used widely.

**Procedure Localization:**

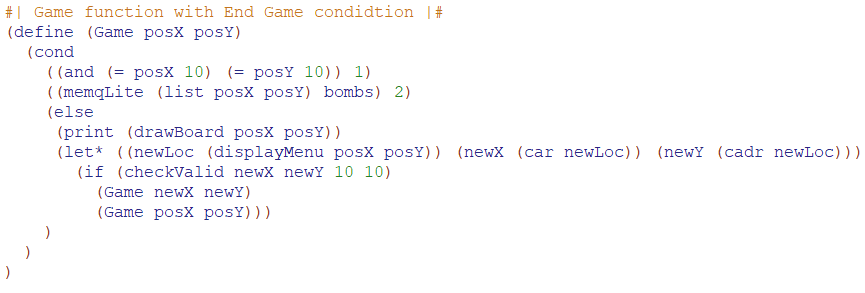
Procedure localization is concept of nesting sub-procedures required for execution within the main enclosing procedure. This allows the sub procedures to utilize variables whose scopes are related to the enclosing procedure with being passed to every sub-procedure.



The maze procedure has row procedure defined within its environment. This allows the row procedure to access all the variables like walls, pacman, location of pacman without being passed parametrically. Thus, row procedure can build the maze row-wise using details of the maze from its enclosing procedure maze.

**Tail Recursion:**

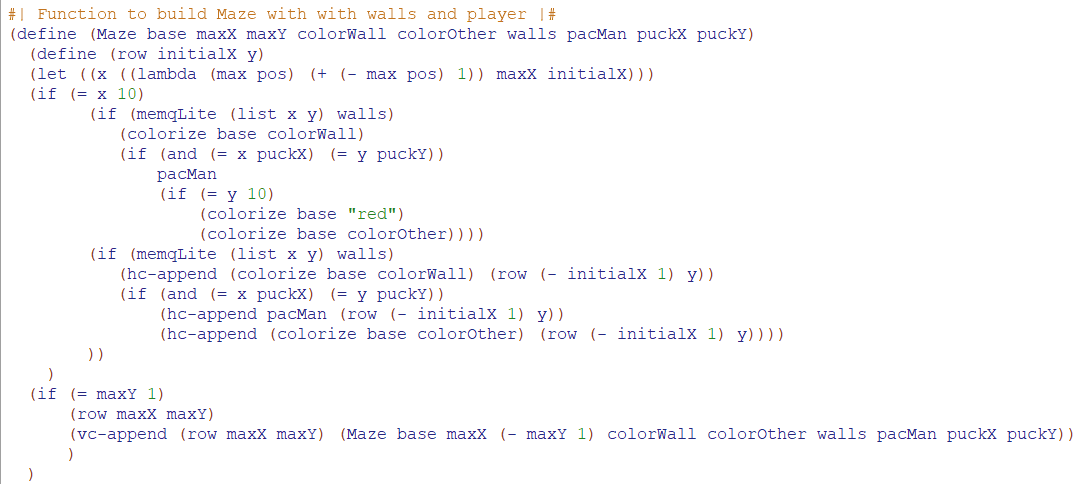
Recursion refers to calling the same procedure within its body to create a looping functionality along with a base condition to stop its progress. Tail recursion calls the same procedure at the end of the procedure body.



Here recursion is used to manage pacman movement across the maze. Till the pacman reaches the destination (or dies when stepping on bombs), new coordinates are calculated at every function call, and if that coordinate is a path (not a wall or bomb), the procedure is called again to simulate the maze from the new coordinates. This continuously happens till the player manages to move the pacman to the destination cell.

**Procedural Abstractions:**

Procedural abstraction allows us to implement procedures with variable parameters. Here, the row function creates each row of the maze with walls, paths, and bombs. Each row is recursively appended together to create the maze.

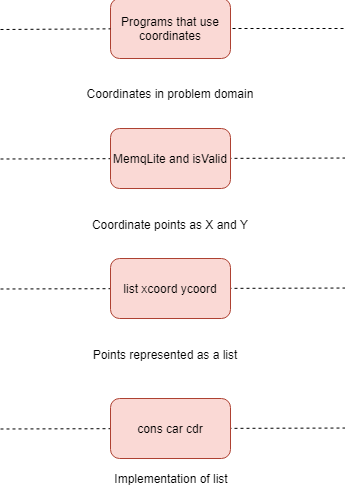


Thus, the procedures such as pacman, base and hc-append serves as an abstraction that creates the entire maze. Here, the row procedure takes in variable parameters in terms of coordinates and based on their type, either a wall or a bomb or a path cell is attached in the row and further developed into a maze.

**Data Abstraction and Abstraction Barrier:**

The coordinates that represent each cell in a maze are defined internally as list of lists and are accessed using car and cdr. This is the data abstraction provided by this maze game. The complex data (coordinates) are internally implemented as list of x and y coordinate pairs (primitive data types). We use various different procedures like memqLite and isValid procedures to access and manipulate the complex data.

Abstraction barriers isolate various levels of a system. At each level of the system, the abstraction barrier separates the programs above that uses the programs defined below that implement the actual data abstraction.

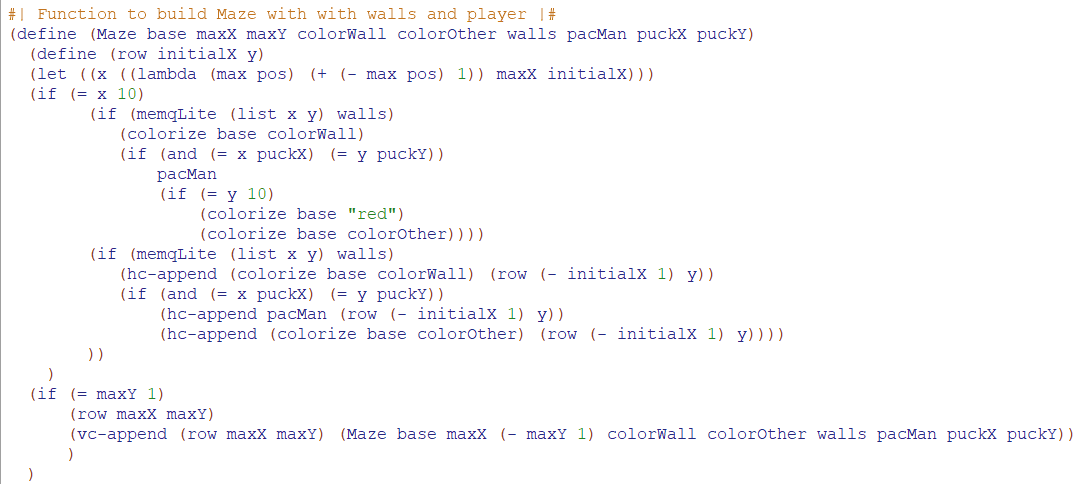


The levels of abstraction and corresponding abstraction barriers are depicted in the above diagram. In a procedural level, the maze game uses coordinate points to generate the entire maze and to track movements. But, to achieve such functionalities, procedures like memqLite and isValid are used. In a more inner level, the coordinate points are defined as list of x and y coordinates. But internally, list in racket is implemented using cons, car and cdr.

As from the above hierarchy of abstraction levels, any program at lower-level can be utilized in the upper-levels. But bypassing abstraction barriers by utilizing upper-level procedures in lower-level procedures should not happen as it induces problems in terms of maintenance and its inner meaning.

**Lexical Scoping:**

Racket is a lexically scoped language. Variables defined in a higher scope of a procedure can still be accessed from within the procedure, but variables defined within this procedure cannot be accessed outside (in the higher scope).



Here, variables defined in the scope of maze procedure is used in its sub-procedure. But variables defined within the lambda procedure cannot be used anywhere outside its scope. Same rule applies to the variables defined in the scope of row procedure.

**Chapter 4** **Gameplay**

