Savannah Nester
CSCI 49201
Functional Programming in OCaml
Professor Nikolaev

OCaml Snake Game Project Report

For my OCaml project, I implemented a 2D game of Snake using only functional programming methods. Snake is a game in which a user controls the movements of a snake with the target to consume as many mice as possible. The game is played with user against the computer, and the player loses if the snake hits one of the four walls along the quadrilateral perimeter or if the snake eats its tail. The user will control the movement of the snake using the arrow keys on their keyboard. I will track the number of points earned by each player in order to determine a final score which will be dependent on the number of mice consumed by the snake, which is also equivalent to the length of the snake's tail. Each time the snake consumes a mouse, its tail will grow in length in order to increase the difficulty of the game as the user progresses.

I used various functions in order to implement this game of snake. The majority of the functionality of this program involved the use of lists and tuples. In order to initialize both the snake and mouse, I created two functions. The create_rat function creates a tuple that serves as a representation of the rat using random numbers so that it can be located anywhere on the board. A value of 1 was added to the random number generated to avoid 0,0 values, and thus the snake being located in the top left corner of the board (which would not be random). The create_snake function places the snake using the same logic, however the create_rat function also checks the location of the mouse to ensure that it is not placed on a location of the board currently occupied by the snake or its tail.

In order to move the snake, a function called get_direction was created. It utilizes the read_line() function in order to get the direction and then match against the type defined. There were four different directions defined by the direction type. These were DIRECTION_UP, DIRECTION_DOWN, DIRECTION_RIGHT, and DIRECTION_LEFT. The characters used to match the direction were ‘s’ for DIRECTION_UP, ‘w’ for DIRECTION_DOWN, ‘a’ for DIRECTION_LEFT, and ‘d’ for DIRECTION_RIGHT. Three functions were created in order to move the snake to a new location on the board. The reverse_list function took the list of tuples that represented the snake, and then returned the reversal of the list, using ‘@’ to concatenate the new tuple. The select_tail function took the list of tuples representing the snake, with each tuple being a pair of values representing that part of the snake’s location, and then returned the tail of the snake. The update_snake_tail function utilized these two functions to reverse the tail and then select the head in order to remove it, and then reversed the list again to maintain the same position for the other tuples of the snake’s tail.

In order to update the game, a game_update function was created. This function updated the state of the game, received the direction to move the snake based on user inputs, and then performed the appropriate actions on the snake list depending on whether a mouse was consumed or not. In order to determine if the mouse consumed a snake, a is_rat_consumed function was created. This checks if the head of the snake and the mouse are located in the same place on the game board. If they are, the space is marked with an ‘X’.

There are two ways in which the player can lose the game and the game ends. The player can either hit one of the walls of the game board, or the user can move the snake into a location occupied by a segment of its tail, thus consuming itself. The hit_wall function checks for the first

way for the game to end. It recursively checks the list representing the snake to determine if the player hit one of the walls of the game board. The consumed_itself function is another recursive function to check if the game has ended. It matches the head of the snake against every segment of its tail to determine if the head and tail segment are located in the same place on the board. If they are, this indicates the snake has consumed itself and the game ends, displaying the user's final score at the end of the game.

There were various challenges I encountered while developing my snake game. The main challenge was utilizing the OCaml graphics library. I spent a great deal of time working on the graphics portion, but unfortunately I was unsuccessful in my implementation of the Graphics library with the fully functional implementation of my code (I created an alternative version of the game in OCaml, but using arrays, in which the graphics library did work, however as this was not a functional implementation I have not included it as part of this project). The end result was limited graphics displayed in the terminal, however it had a fully functional implementation using lists and tuples. Prior to submission of this report, and after presentation of my project in class, I worked on getting the graphics console to work with my code. I was able to get it to work once at the start of the game, however once I moved the snake into a mouse location, the graphics window would close and I would not be able to access the game. I am still working on fixing this bug, and although it will be after the project deadline, I would like to get it to work so that I can play the game, and perhaps share it with others.

In conclusion, this project helped me to solidify my understanding of both the OCaml language and functional programming practices. Though I encountered several challenges, and though my game is incomplete as it is missing the working graphics library functions,

implementing the logic of the snake game and refining my understanding of OCaml are two accomplishments I feel I can take away from this project.