CSC330 Summer 2004 Assignment 4

George Tzanetakis

July 17, 2004

1 Overview

This assignment will be done in groups of two. You are free to choose your own group. In case you do, I will need an email from both partners to confirm that they are in group. Otherwise you will be assigned randomly a partner. YOU NEED TO EMAIL ME IN EITHER CASE. FAILURE TO DO SO MIGHT RESULT IN NOT GETTING GRADE FOR THE ASSIGNMENT.

The careful design and documentation (i.e comments) of your code will be important factors in your grade. If there is a bug it's better to report it than hide it. Be honest, precise and clear and you shall be rewarded. NEVER (at least in this class) sacrifice clarity for efficiency of execution unless explicitely asked to do so. PROVIDE A SET OF WELL-DESIGNED TEST CASES FOR EVERY PART OF THE ASSIGNMENT.

The assignment is worth 10 points (10% of the final grade) and will be graded in using half-point intervals. The overall documentation and packaging of your submission according to the instructions provided in the web page (READ THEM CAREFULLY) will be worth 1 point.

The main goal of the assignment is to familiarize with how modules and objects can be used to structure programs. It is also a good exercise to learn more about ML and Smalltalk. I hope you find the assignment interesting and helpful.

2 Part 1 (4 pt)

In this part we will use a functor to implement generic matrix arithmetic. To simplify things we will only consider zero and sum as the operations that the elements of a matrix must support. This can be expressed using the following signature:

```
signature ZS =
sig
    type t
    val zero : t
    val sum  : t * t -> t
end;
```

An example of structure implementing the above signature is IntZS.

```
structure IntZS:ZS =
struct
    type t = int
    val zero = (0)
    fun sum(x,y) :t = x + y;
end;
```

Similarly implement the following structures that satisfy the ZS signature: RealZS, ComplexZS and RationalZS (2pt)

Use tuples of reals to represent complex numbers and tuples of integers for rational numbers. For example the complex number $1 + i$ will be represented as (1.0, 1.0). To add two complex numbers the we add the real parts and imaginary parts. So if a is $1+i$ and b is 2 then a+b is $3+i$ which would be (3.0, 1.0).

For example the rational number $\frac{1}{2}$ will be represented as (1,2). The following rule can be used to add rational numbers $\frac{a}{b} + \frac{c}{d} = \frac{a*d+b*c}{b*d}$

The rational numbers need to be reduced to their lowest term representation after each operation by dividing both nominator and denominator with their greatest common divisor. For example $\frac{10}{12}$ needs to be reduced to $\frac{5}{6}$.

```
val a = (0.0, 1.0)
val b = (0.3, 0.0)
val c = ComplexZS.sum(a,b)
val a = (1,2)
val b = (1,3)
val c = RationalZS.sum(a,b)
```

The answers should be c = (0.3,1.0) for the complex numbers and c = (5,6) for the rationals.

Implement a functor that can be used to generate specific structure for matrix arithmetic.(2pt)

Here is a rough outline of how it will look/be used. Matrices are represented as lists of rows (each row in turn is a list of elements Z).

```
functor MatrixZS(Z:ZS): ZS =
struct
   type t = Z.t list list;

   val zero = [];

   (* write sum here using only ListPair.map and Z.sum *)
end;

structure IntMatrix = MatrixZS(IntZS);
IntMatrix.sum([[1,2], [3,4]], [[5,6],[7,8]]);
(* the answer should be: val it = [[6,8],[10,12]] : IntMatrix.t *)

structure RationalMatrix = MatrixZS(RationalZS);
RationalMatrix.sum([[(1,2),(1,2)], [(3,4),(3,4)]], [[(1,3),(1,2)], [(1,2),(1,3)]]);
(* the answer should be: val it = [[(5,6),(1,1)],[(5,4),(13,12)]] :
RationalMatrix.t *)
```

3 Game State Management (6pt)

As we talked about in class, one of the major strengths of object-oriented programming is the ability to simulate/model the real world. In this part of the assignment you will implement a simple but not trivial backend to a multiplayer/multisession adventure game. We will not deal with any graphics, storylines, combat-systems, sockets/networks etc (these are left for extra credit).

The main goal of this part is to familiarize you with state-management using the imperative object-oriented programming style and teach you more about Smalltalk.

The game is relatively simple but has a few interesting constraints. Multiple games can be played simultaneously. All aspects of the game: players, items, rooms can be added while the game is running.

Here are the basics operations that need to be supported. Items have a name and a value in coins. Players can pick and drop items. The value of players is increased everytime they pick an item and decreased everytime they drop an item correspondingly based on the value of the item.

Rooms are connected to each other by exits. Each room can have an arbitrary number of exits each labeled by a unique identifier. For example the traditional #north, #south, #east, #west movement can be supported with appropriate commands but also other exit names such as #teleport, #george or whatever else you image. There is no limit to the number of exits a room can have and exits are directed (go only in one direction). One can think of the map of the world as a directed graph with each edge labeled by the exitName. So to have proper north/south movements two edges need to be created: one for the north direction and one for the South.

Players can move from room to room following the exits and the game must keep track of where they are. Obviously dropping and picking items is based on the room the player is in.

Although simple this game shows the foundations behind any multi-user turn-based adventure game.

3.1 SmallTalk implementation

You will need to create the following classes (more details are provided later):

- *GameObject*: Player and Item are derived classes of this class which serves as an interface. Use the subclassResponsibility keyword to delegate the implementation to the derived classes. Read about it in your book.

- *Item*: A derived class from *GameObject*. The main method to implement is print which just outputs the name of the item and it's value in coins.

- *Player*: A derived class from GameObject. The main method to implement is print which just outputs the name of the player and it's value in coins followed by a list of *Items* the player is carrying. A Player contains a *Dictionary* of *Items*. In addition to supporting print it supports the following messages: pickItem:, dropItem:, lookupItem:, hasItem: with the obvious semantics.

- *Room*: has a name, a dictionary of neighbors (other *Rooms* and a dictionary of *GameObjects* (players or items). It supports the following messages: addNeighbor:, getNeighbor, removeNeighbor:, addObject:, removeObject:, lookupObject:. Print a room shows the name, objects in side it (players or items) and the available exits.

- *Game:* this is the main class for the game. Individual game sessions are instances of the Game class. The following messages are supported: addPlayer:, addRoom:, addItem:, connect:, move:, pick:, drop:, lookThrough:. A dictionary of playerPositions associating players with rooms and a dictionary of rooms will probably be needed.

The game initially starts with one room the #ReceptionHall but new rooms can be added at runtime. Here is an example session with the game:

```
;; Start g1 game session
(val g1 (init Game))

;; add two players
(addPlayer: g1 #Dale  100)

;; see the game through the eyes of Dale
;; important when each player is in a different room
(lookThrough: g1 #Dale)
(addPlayer: g1 #David  50)
(lookThrough: g1 #David)

;; add an items
(addItem: g1 #ReceptionHall #Midterm 15)
(lookThrough: g1 #David)
(addItem: g1 #ReceptionHall #Final 40)

;; pick and drop items
(pick: g1 #David #Midterm)
(pick: g1 #Dale  #Final)
(drop:  g1 #Dale  #Final)


;; Game 2 player and item commands
(val g2 (init Game))
(addPlayer: g2 #George  100)
(lookThrough: g2 #George)
(addItem: g2 #ReceptionHall #Midterm 15)
(addItem: g2 #ReceptionHall #Final 40)
(pick: g2 #George #Midterm)
(pick: g2 #George #Final)
```

Here is the output before Dale drops the final:

```
Location=ReceptionHall
Visible=
Player=Dale-140
Carrying=
___Item=Final-40
_
Player=David-65
Carrying=
___Item=Midterm-15
```

and here is after Dale drops the final:

```
Location=ReceptionHall
Visible=
Player=Dale-100
Player=David-65
Carrying=
___Item=Midterm-15
_
Item=Final-40
```

The exact formating of the output is up to you as long as the proper semantics of drop/pick and the other commands are implemented. Now let's create some rooms and move around.

```
;; Building a map (each room can
;; have an arbitrary number of unidirectional exits)
(addRoom: g1 #NorthRoom)
(addRoom: g1 #Bar)

;; make directed edges/exits between rooms
(connect: g1 #ReceptionHall #north #NorthRoom)
(connect: g1 #NorthRoom #south #ReceptionHall)
(connect: g1 #ReceptionHall #drink #Bar)
```

```
(connect: g1 #Bar #return #ReceptionHall)

;; addItem in a new room
(addItem: g1 #Bar #Beer 20)

;; Movement + drop/pick
(move: g1 #Dale #north)
(move: g1 #David #drink)
(drop: g1 #David #Midterm)
(pick: g1 #David #Beer)
(lookThrough: g1 #David)
(drop: g1 #David #Beer)
(lookThrough: g1 #David)
(pick: g1 #David #Beer)
(move: g1 #David #return)
(lookThrough: g1 #David)
```

Here is the output of the last lookThrough command.

```
Location=ReceptionHall
Visible=
Item=Final-40
Player=David-70
Carrying=
___Item=Beer-20
-
Exits=
north
drink
```

It is my hope that these examples clearly show the semantics of the game. If you have any further questions let me know via email.

IMPORTANT: YOU CAN ASSUME EACH ROOM, PLAYER, ITEM HAS A UNIQUE NAME

IMPORTANT: THE PURPOSE OF THIS PART IS TO TEACH YOU ABOUT OBJECT ORIENTED PROGRAMMING. IT IS POSSIBLE TO CREATE ONE BIG OBJECT THAT DOES EVERYTHING BUT THAT IS NOT WHAT I WANT. MAKE SURE YOU FOLLOW THE DIRECTIONS FOR STRUCTURING THE CODE INTO OBJECTS.

3.2 Deliverables

The code for the game MUST be written using the Smalltalk-like language interpreter provided (the interpreter is written in ML). You will notice that the existing implementation does not directly support deleting a key,value association from a dictionary. To get around this you will either need to modify the initial basis code in the interpreter or make your own extension of the Dictionary class. In either case you will need to understand how Dictionaries are implemented. After that the rest of the assignment is pretty much manipulating the dictionaries appropriately.

Grading is straightforward 3 points for addPlayer, addItem:, pick:, drop: (single room mode) and 3 points for being able to add rooms, move (multi room mode).

4 Extra Credit

- (2pt) Extend the signatures, structures and functor of Part 1 to have the full albebraic specifications (sum, diff, prod, quot, one, zero). Make matrices where each element is matrix of real numbers.
- (4pt) Implement pretty printing for the all the structures defined in part 1. Write a function toLatex() that converts each structure to the appropriate latex commands to print the stuff nicely like in math book. Part of the extra credit is learning latex and how to use it.
- (1pt) Implement part 2 (the game engine) in actual Smalltalk. Any implementation (Squeak, GNU) is fine.
- (1pt) Implement part 2 in C++. Write a 1 page paper contrasting the two implementations.
- (4pt) Make an interesting game in actual Smalltalk based on the above code. For example add network support, graphics, map generators etc. Any reasonable addition will count however grading is up to me with a maximum of 4 points.

5 Submission

Please follow the submission guidelines from the course webpage. In summary tar and gzip everything into one file. Include a README explaining how things are structured. Include all the code necessary to compile the assignment not just the parts you extended/modified.

HAVE FUN AND I HOPE YOU ENJOY THIS ASSIGNMENT AS MUCH AS I ENJOYED PREPARING IT