PLT Language Reference Manual

Lawrence Chillrud (lgc2139), Jack Damon (jad2267),
Richa Gode (rg3006), and Sagar Lal (sl3946)

March 6th, 2019

Grid over Time, or GoT, is a language that is designed to show how a designated space evolves over time. The language allows for many standard data types, as well as grid data structures and object data structures. Users will be able to construct a discrete canvas and then dictate how the different objects should move and change shape in the space over each time interval. The goal of the language is to allow for the easy implementation of rules for agents, which will allow for many different types of creations.

1 Lexical Elements

1.1 Identifiers

Identifiers are used to label agents, grids, and other data types. They can begin with upper or lower case characters, but any multi-word identifiers must be split using an underscore, for example The_knights_of_the_round_table.

1.2 Reserved keywords

int	float	char
bool	string	main
while	in	for
if	else	elif
and	or	new
new	def	class
self	return	true
false	**@step**	#

1.3 Literals

- **Integers**: Any sequence of digits
- **Floats**: Any sequence of digits with a decimal point
- **Bools**: either true or false

- **Chars**: Composed of a single ASCII character surrounded by single quotes
- **Strings**: Composed of several ASCII characters surrounded by double quotes

1.4 Operators

Operator	Description
+, -	addition and subtraction
*, /, %	Multiplication, Division, and Modulo
^	exponent
>, >=, <, <=	inequality operators
==, !=	equal and not equal
!	not symbol
=	assignment
&&, ||	logical and/or
.	access

1.5 Delimiters

- **Parentheses** group expressions to force precedence as well as enclosing function arguments
- **Commas** Separate function arguments
- **Semicolons** End statements
- **Curly braces** Enclose function definitions, conditional blocks, loops, and object definitions

1.6 Whitespace

Only used to separate tokens

1.7 Comments

is used to indicate a single line comment

/# begins a commented out block, while #/ ends a comment block

2 Data Types

2.1 Primitive Data Types

int: 32-bit signed integer value

```
int x = 42
```

float: 32-bit floating point value

```
float x = 4.2
```

bool: True or False

```
bool is_happy = True
```

2.2 Non-primitive Data Types

1. Tuple
A *tuple* is a built in datatype composed of n items that do not have to be of the same type.
Syntax:

```
(<object_1>, <object_2, ..., <object_n>)
```

Example:

```
(Knight_1, (4,4))
```

2. List
A *list* is a built in datatype composed of multiple values of the same type.
Syntax:

```
List <name> = [<list_obj_1>, <list_obj_2>, ..., <list_obj_n>]
```

Example:

```
List my_knights = [Knight_1, Knight_2, Knight_3]
```

Operations:

`List.append(<list_obj>)` adds `<list_obj>` to the end of the list

`List[i]` returns `<list_obj>` at index i, or an error if i is out of bounds

`List.delete(i)` deletes the object at index i and readjusts the indexes of trailing list items. Returns an error if i is out of bounds.

3. Agent
An *agent* is a built in abstract datatype that represents the objects that will interact on the grid.
Syntax:
Abstract extension:

```
def agent <agent_name> = {
   color = (x, y, z) # an RGB color value
   def move(){
      # the move function will be called at the end of each time
step for each agent
      # user must return an (x, y) tuple indicating the movement
to be made in each direction
   }
   def interact(){
      # the interact function will also be called at the end of
each time step for each
      # the user can define custom behavior in this function
hook.  There is no expected return value
   }
}
```

Example:

```
def agent Knight = {
   color = (169, 169, 169) # grey
   def move(){
      direction = rand(1, 4)
      if(direction == 1){
         move_left();
      }
      elif(direction == 2){
         move_right();
      }
      elif(direction == 3){
         move_down();
      }
      elif(direction == 4){
         move_up();
   }
   def interact(){
      # custom interaction code
   }
}
# Agent-extension instantiation example
Knight knight_1 = new Knight();
```

Operations:

`agent.get_location()` returns an $(x,\ y)$ tuple representing the location of the agent at the current time step.

`agent.move()` Updates the agents location to tuple (*x-1*, *y*) if this is within grid bounds.

`agent.move_right()` Updates the agents location to tuple (*x+1*, *y*) if this is within grid bounds.

`agent.move_down()` Updates the agents location to tuple (*x*, *y+1*) if this is within grid bounds.

`agent.move_up()` Updates the agents location to tuple (*x*, *y-1*) if this is within grid bounds.

4. Grid
A *grid* is a built in datatype representing the "board" on which agents exist, move, and interact with one another.
Syntax:

```
grid <grid_name> = new grid((grid_height, grid_width), hasTime)
```

Example:

```
grid my_timeless_grid = new grid((100, 100), False)
```

Operations:
`grid.introduce(<agent_instant_name>, (<x_spawn_location>, <y_spawn_location>))`
Built in function that instantiates `<agent_instant_name>` (an extended-agent instance) at the coordinate location of the passed in (*x*, *y*) tuple

`grid.start(<time_steps>)`
Built in function that takes an int `<time_steps>` that specifies the amount of steps this grid should be "run" (in other words, this determines how many times each agents move and interact function should be called).

3 Functions

3.1 Built in Functions

The following functions are built in functions to the GOT language:

```
rand(x, y):  takes a lower bound int x and upper bound int y
and returns a randomly generated integer between the two
(including x and y).
```

3.2 User Defined Functions

Users can declare functions using the following syntax:

1. **Function with arguments**

```
def <function_name>(<arg_1>, <arg_2>, ..., <arg_n>){ ... }
```

2. **Function without arguments**

```
def <function_name>(){ ... }
```

Users can choose whether the function returns something or not. Return types do not have to be declared.

4 Control Flow

The following keywords are reserved for control flow: `while, for... in, for each..., if... elif (optional)... else (optional)`.

1. **While Loops:**

```
while (bool-expression) { do a; }
```

2. **Conditional Loops:**

```
if (bool-expression) { do x; }
elif (bool-expression) { do y; }
else { do z; }
```

5 Program Structure and Scope

A user executes a program in our language in **one file**. The file executes a **main function** at the bottom, which instantiates the grid with given dimensions, and introduces the different characters that a user would want to place on the board. Then, the grid is started using the syntax described in an earlier section.
sample main function:

```
def main(): {
   grid camelot = new grid((100, 100), true);
   camelot.start(100); }
```

The scope is restricted within logical blocks (entities, functions, etc) by **curly braces**. Within a certain block, variables can be referenced by the local variable **self**.

6 Sample Program

```
/# DEFINING AGENTS #/
def agent Rock = {

    int lifepoints;

    def set_lifepoints(lifepoints_setter){
        lifepoints = lifepoints_setter;
    }

    def get_lifepoints(){
        return lifepoints;
    }

    color = (100,100,50); #The color is described as a tuple of RGB values.

    /# Move requires a function that returns a loc (x,y) pair
     # indicating movement on the grid.
     # E.g., returning (-1, 4) would move the agent
     # backwards along the x-axis 1 square, and
     #/ upwards along the y axis 4 squares.
    def move() {
        return (0,0); #it doesn't move - it's a rock
    }

    def interact() {
        #Checks if rock is still 'alive' every turn
        if (self.get_lifepoints() < 0) {
            self.destroy(); #'self.destroy()' is supplied by the language for all agent objects
        }
    }

}

def agent Knight = {

    int lifepoints;
    int damage;
    int food;
    bool move = true;

    def set_lifepoints(lifepoints_setter) {
        lifepoints = lifepoints_setter;
    }

    def get_lifepoints() {
        return lifepoints;
    }

    def set_food(food_setter) {
        food = food_setter;
    }
```

```
def get_food() {
    return food;
}

/#
# Matrix constructor is of form Matrix(height,
# width, left-to-right, top-down array in which 0
# means an 'off' and a color indicates an 'on'
# pixel of that specified color)
#/
color = (50,60,80);

def move() {

    # if/else to slow down Knight movement...
    # only have it move every other turn

    if(move){
        int x = rand(-5, 5);
        int y = rand(-5, 5);
        move = false;
        return (x, y);
    }
    else {
        move = true;
        return (0,0);
    }

}

def interact() {

    loc = self.Location;

    for agent in loc.Agents {
        if (agent.Type == "Thief") {

            int new_lifepoints = agent.get_lifepoints - self.damage;

            agent.set_lifepoints(new_lifepoints); #the knight does damage to Thief
                                                  #that is at the same location


            if (new_lifepoints <= 0) {
                food_from_agent = agent.get_food();
                my_food = self.get_food();
                self.set_food(my_food + food_from_agent);
            }
        }

    }

    if (self.food == 0) {
        self.lifepoints = self.lifepoints - 10;
```

```
        }

        if (self.get_lifepoints() < 0) {
            self.destroy(); #'self.destroy()' is
            #supplied by the language for all agent objects
        }


    }

}

def agent Thief = {

    int lifepoints;
    int food;
    int damage;

    def set_lifepoints(lifepoints_setter) {
        lifepoints = lifepoints_setter;
    }

    def get_lifepoints() {
        return lifepoints;
    }

    def set_food(food_setter) {
        food = food_setter;
    }

    def get_food() {
        return food;
    }

    color = (250,0,0);

    def move() {
        int x = rand(-2, 2);
        int y = rand(-3, 3);
    }

    def interact() {

        loc = self.Location;

        for agent in loc.Agents {
            if (agent.Type != self.Type) {
                food_to_steal = agent.get_food();
                if(food_to_steal <=5){
                    agent.set_food(0);
                    self.set_food(self.food + food_to_steal);
                }
                else {
                    agent.set_food(food_to_steal - 5);
```

```
                    self.set_food(self.food + 5);
                }
                agent_lifepoints = agent.get_lifepoints();
                agent.set_lifepoints(agent_lifepoints - self.damage);
            }
        }

        if (self.food == 0) {
            self.lifepoints = self.lifepoints - 10;
        }

        if (self.get_lifepoints() < 0) {
            self.destroy(); #'self.destroy()' is supplied by the language for all agent objects
        }

    }

}

def agent Farmer = {

    int lifepoints;
    int food;

    def set_lifepoints(lifepoints_setter) {
        lifepoints = lifepoints_setter;
    }

    def get_lifepoints() {
        return lifepoints;
    }

    def set_food(food_setter) {
        food = food_setter;
    }

    def get_food() {
        return food;
    }

    color = (180,90,2);

    def move() {
        int x = rand(-1, 1);
        int y = rand(-1, 1);
    }

    def interact() {

        # farmer 'farms' 5 food units every turn
        self.set_food(self.get_food() + 5);

        loc = self.Location;
```

```
        for agent in loc.Agents {
            if (agent.Type == "Knight" and (self.get_food() > 5)) {
                int my_food = self.get_food();
                self.set_food(my_food - 5);
                int agent_food = agent.get_food();
                agent.set_food(agent_food + 5);
            }
        }

        if (self.get_lifepoints() < 0) {
            self.destroy(); #'self.destroy()' is supplied by the
            # language for all agent objects
        }

    }

}

/# END OF AGENTS #/

def main {
    # grid constructor is of form grid((height, width), has_time)

    grid narnia = new grid((100, 100), true);

    /# using grid.introduce() function in form of
     # grid.introduce((Agent, number_of_agents_to_add, location), ...)
     # Can also add individual agent instances to
     # specific locations eg: grid.introduce((agent, (x, y)))
     #/

    Rock rock_1 = new Rock();

    Thief thief_2 = new Thief();

    Knight knight_3 = new Knight();

    Farmer farmer_4 = new Farmer();

    narnia.introduce(rock_1,(2,3));

    narnia.introduce(thief_2, (4,10));

    narnia.introduce(knight_3, (6,8));

    narnia.introduce(farmer_4, (9,30));

    /# takes how many 'time steps' to run. A tie step
     # consists of calling the move()
     # function for each agent followed by the interact() function for each
     # agent (after each agent has moved)
     #/
    narnia.start(100);
}
```