tiler.

Manager: Jason Lei (jl3825)
Language Guru: Monica Ting (mst2138)
System Architect: Evan Ziebart (erz2109)
System Architect: Jiayin Tang (jt2823)
Tester: Jacky Cheung (jc4316)

Language Description:

tiler is a language designed to simplify the process of programming turn-based, *square* tiled games. Tiles are basic descriptive units of the game environment and are arranged in a grid. tiler allows programmers to interface with tiles through a coordinate system. Tile-based board games (*Monopoly*, *Chutes and Ladders*, *Sorry!*, Chess, Checkers, etc.) lend themselves easily to be programmed by tiler. However, tiler can handle more ambitious games, specifically tactical RPGs (*Final Fantasy Tactics*, *Fire Emblem*, *Advanced Wars*, etc).

In addition to allowing the programmer easily create and interface with the game's environment, tiler allows the programming of objects (such as boardgame pieces or video game characters) and to assign these objects attributes that signify its roles in the game. This allows for easy control of objects with specific attributes, thereby reducing code redundancy and providing scalable control of objects of identical/similar types. Furthermore, the coordinate system allows for intuitive control of movement of objects during each discrete turn and tiler provides operations to easily define movements that are and are not allowed.

To further simplify the programming of games, tiler is structured around code blocks (described in depth under "Language Concepts"). These blocks represents aspects of games that are universal (grid/board initialization, gameplay, rules, interactions, etc). The abstraction of these concepts allows for improved code readability compared to conventional programming languages and reduces the learning curve required to understand the foundations needed to create a game. In addition, a variety of built-in game-specific functions allow for games to be expressed in a less verbose manner; this also improves code readability and reduces programming time.

tiler's syntax is designed to be both familiar for experienced programmers and simple enough for beginner programmers seeking to program games to easily pick up. This simplicity stems from the reduction in abstraction of game elements required when programming games in more general-use languages. tiler can feasibly be used to program for other purposes, but its structure and syntax make it intuitive to program turn-based tile games and do not lend itself easily to other purposes.

Language Concepts:

Board

The programmer operates within the framework of a board consisting of 2D grids. The board simultaneously acts as a windowed environment for the program and a coordinate system for gameplay.

Objects

The concept of objects corresponds to the pieces which occupy the game board. Each object consists of a series of attributes which define its type and behavior. All objects have at minimum an attribute which defines their current location on the board grid.

Variables

Behave much like in traditional programming languages. Can be declared in the init code block and then scope to any other code block of the program.

Attributes

Attributes define the type and behavior of the objects on the board. They act similarly to variables in that they can take on various values, called states. However, unlike variables they are specific to objects. The programmer typically enumerates these states when declaring attributes for an object.

Turn

Every game is made up of basic units called turns, during which the program will behave the same on each looping. There may be several turns in the game which execute in sequence to add up to a round. The round is then repeated until the game ends.

Code Blocks

In the tiler language, code is divided into different labeled code blocks, each serving a specific function in defining the larger game. Each one is attached to a specific keyword in the language. A quick run-down of these code blocks is as follows:

- **init**: used by the programmer to define how the game should be at the start of execution. This code serves the function of setting up the board as well as initial placement and states for all the objects involved in the game. Additionally, this part of the code is where the programmer defines variables which they would like to use in other blocks.

- **turn**: in these code blocks, the programmer defines how the game behaves during each of the one or more turns. The game might prompt the player, listen for input, and/or manipulate the entities on the board. This allows the programmer to abstract away the need for a “game loop”, since these code segments will loop continuously during play.

- **end**: this block is where the programmer defines a series of conditions which indicate the game has reached a final state.This is done using the keyword case to specify each of the conditions. It is also where the programmer defines game behavior once the end conditions are met

- **rules**: this is the part of the code where the programmer specifies object behavior during gameplay, usually as dictated by the object's attributes. The programmer can then specify change of state for important object attributes such as coordinates on the board.

- **class**: the programmer uses this part of the code to define the objects that will populate the board during game execution. In particular, attributes for the objects are specified, as well as the states they can take on.

Design Principles:

Datatypes

- Primitives: int, float, string, boolean, null
- Arrays
  - Limited to 1D arrays of any types listed in (a)
  - A programmer would not have to manipulate 2D matrices themselves
- Map
  - In curly braces, separated by commas
  - `{1: 'one', 2: 'two'}`
- Tuples
  - `(x, y)`
  - Coordinates

Operators

- `+ - * / %`
- `== != ! && || > < >= <=`
- `>> !>>`
  - 'can be moved to' or 'can be placed at' and its negation
  - `if (rook >> (2, 2)) { … }`
- `// single line comment`
- ```
  /*
    multi-line
    comment
  */
  ```
- `'string'`
  ```

Keywords and Built-in Functions

- Block keywords: init, rules, turn, end
- Move
  - `move((x, y)), move(left, 1)`
  - object method that changes the coordinate of the piece based on arguments that are passed into it
- Random
  - `random()`: generates a random float between 0 and 1
  - `random(board)`: generates a random coordinate in the board/grid
  - `move(random(board))`
- Miscellaneous
  - `grid(int x, int y)`: creates a board with dimensions x by y
  - `background(string filename)`: sets the background image for the board
  - `captureMouse()`
  - `print(string message)`
  - `prompt()`: get user input, returns string

Sample Code:

Below are 3 example programs for tic-tac-toe, chess, and an obstacle game.
Tic-tac-toe is a simple game with several possible winning conditions.
Chess contains a sample rule block.
The obstacle game shows how multiple object classes can be created to have different attributes.

```
/* create your own "objects" by defining them as a new class
   each new object class implicitly contains x and y coordinates (ints)
   each new object class implicitly contains get/set methods for coordinates
   programmer can define their own additional attributes
   programmer can define their own additional methods
*/

// tic tac toe

class Piece {
   // define additional attributes
   attr players: string {'x', 'o'};
}

init {
   // set board size and optional background image
   board(3, 3);
   background('3x3board.jpg');
   int x, y;
}
```

```
turn player1 {
   print('x turn: click an empty tile');
   // keep capturing mouse coordinates until it captures a valid (empty) tile
   do {
      x, y = captureMouse();
   } while (!board[x,y].isEmpty());
   board[x,y] = new Piece(player='x');
}

turn player2 {
   print('o turn: click an empty tile');
   do {
      x, y = captureMouse();
   } while (!board[x,y].isEmpty());
   board[x,y] = new Piece(player='o');
}

end {
   case 1: for tile in board.row[0] {
      if (tile.player == board[0,0].player) {
         print(board[0,0].player + ' wins');
      }
   }

   case 2: for tile in board.row[1] {
      if (tile.player == board[0,1].player) {
         print(board[0,1].player + ' wins');
      }
   }

   case 3: for tile in board.row[2] {
      if (tile.player == board[0,2].player) {
         print(board[0,2].player + ' wins');
      }
   }

   case 4: for tile in board.column[0] {
      if (tile.player == board[0,0].player) {
         print(board[0,0].player + ' wins');
      }
   }

   case 5: for tile in board.column[1] {
      if (tile.player == board[1,0].player) {
         print(board[1,0].player + ' wins');
      }
   }

   case 6: for tile in board.row[2] {
      if (tile.player == board[2,0].player) {
         print(board[2,0].player + ' wins');
      }
   }
```

```
   case 7: if (board[0,0].player == board[1,1].player
            && board[0,0].player == board[2,2].player) {
      print(board[0,0].player + ' wins');
   }

   case 8: if (board[0,2].player == board[1,1].player
            && board[2,0].player == board[1,1].player) {
      print(board[1,1].player + ' wins');
   }

   bool gameFull = True;
   for tile in board {
      if (tile.isEmpty) {
         gameFull = False;
      }
   }

   case 9: if (gameFull) {
      print('Tie');
   }
}
```

// chess

```
class Piece {
   attr player: string {'black', 'white'};
   attr role: string {'pawn', 'knight', 'bishop', 'rook', 'queen', 'king'};
   attr removed: bool {True, False}
}

init {
   board(8, 8);
   background('chessboard.jpg');
   int x, int y;

   // place objectss on the board
   for tile in board.row[1] {
      tile = new Piece('black', 'pawn', False);
   }
   board[0,0] = new Piece('black', 'rook', False);
   board[0,7] = new Piece('black', 'rook', False);
   board[0,1] = new Piece('black', 'knight', False);
   board[0,6] = new Piece('black', 'knight', False);
   board[0,2] = new Piece('black', 'bishop', False);
   board[0,5] = new Piece('black', 'bishop', False);
   board[0,3] = new Piece('black', 'queen', False);
   board[0,4] = new Piece('black', 'king', False);

   for tile in board.row[6] {
      tile = new Piece('white', 'pawn', False)
   }
```

```
   board[0,0] = new Piece('white', 'rook', False);
   board[0,7] = new Piece('white', 'rook', False);
   board[0,1] = new Piece('white', 'knight', False);
   board[0,6] = new Piece('white', 'knight', False);
   board[0,2] = new Piece('white', 'bishop', False);
   board[0,5] = new Piece('white', 'bishop', False);
   board[0,3] = new Piece('white', 'queen', False);
   board[0,4] = new Piece('white', 'king', False);

}

rules {
   Piece(player='white', role='pawn'): [i, j] >> {[i, j+1]}
   Piece(player='black', role='pawn'): [i, j] >> {[i, j-1]}
   Piece(player='knight'): [i, j] >> {[i+1, j+2], [i-1, j-2], [i-1, j+2],
      [i+1, j-2], [i+2, j+1], [i-2, j-1], [i+2, j-1], [i-2, j+1]}
      .
      .
      .
}

turn white {
   print('White's turn: select a piece');

   do {
      x, y = captureMouse();
   } while(board[x,y].player == 'white');

   Piece piece = board[x,y];

   print('Select a tile to move to');
   do {
      x, y = captureMouse();
   } while(rule[piece]);

   if (!board.isEmpty(x, y)) {
      board[x,y].removePiece();
   }
   piece.move(x, y);
}

turn black {
   print('Black's turn: select a piece');

   do {
      x, y = captureMouse();
   } while(board[x,y].player == 'black');

   Piece piece = board[x][y];

   print('Select a tile to move to');
   do {
      x, y = captureMouse();
   } while(rule[piece]);
```

```
   if (!board.isEmpty(x, y)) {
      board[x,y].removePiece();
   }
   piece.move(x, y);
}

end {
   int kingCount = 0;
   string winner;
   for tile in board {
      if (tile.piece.role == 'king') {
         kingCount++;
         winner = tile.piece.player;
      }
   }
   if (kingCount == 1) {
      print(winner + ' wins');
   }
}
```

// obstacle/maze game

```
// example of creating multiple classes for game
class Character {
   attr strength: int;
   attr name: string;
}

class Obstacle {

}

...
```