tiler.

Language Reference Manual

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

Table of Contents

1 Program Structure 3

1.1 Code Blocks

1.2 Game Loop

2 Lexical Elements 6

2.1 Identifiers

2.2 Reserved Keywords

2.3 Literals

2.4 Delimiters

2.5 Whitespace

2.6 Comments

2.7 Operators

2.8 Control Flow

3 Datatypes 9

3.1 Primitive Types

3.2 Non-primitive Types

3.3 Variable Declarations

3.4 Classes

4 Functions 13

4.1 User-Defined Functions

4.2 Built-In Functions

4.3 Random Generator Functions

4.4 Object Specific Functions

5 Sample Code 15

1 Program Structure

1.1 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 code block is indicated by a specific keyword in the language, and the scope of each block is marked by a pair of curly braces.

1.1.1 class

Defines the objects that will populate the board during game execution. Additional attributes for the objects and the states they can take on are specified by the programmer

```
class Piece {
   // define additional attributes
   attr player: string ['black', 'white'];
   attr role: string ['pawn', 'knight', 'bishop', 'queen', 'king'];
}
```

Each instance of a class also implicitly contains the following attributes and default values:

```
attr className: 'Piece';
attr x: null;
attr y: null;
attr coord: (null, null);
attr image: null;
```

1.1.2 rules

Associated with a single class and specifies object behavior during gameplay, as dictated by the object's attributes. The rules block does two things:

1. Defines all of the possible moves for an instance of the object, based on its current state. If multiple rule conditions are satisfied, the set of possible moves is the union of the possible moves for each satisfied rule.

```
rules Piece {
   player: 'white' && role: 'pawn' >> [(x, y+1)];
   player: 'black' && role: 'pawn' >> [(x, y-1)];
}
```

2. Acts as a function that can be called with the object as an implicit argument and a tuple as an explicit argument. Returns True if the object can move to the next coordinate and False otherwise. If a rules block is not defined for a class, then the rules function by default returns false.

```
piece.rules(nextCoord)
```

1.1.3 init

Defines 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.

```
init {
   grid(8, 8);                     // set grid size
   background('chess.jpg');        // set optional background image
   int x, y;                       // initialize global variables
}
```

1.1.4 turn

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.

```
turn {
   // developer code here
}
```

1.1.5 end

Defines a series of conditions which indicate the game has reached a final state. If the end block returns, the game is over. If nothing is returned, the game loop continues on to the next turn.

```
end {
   if (endCondition) {
      return 'player1'
   }
}
```

1.1.6 function

Define reusable functions. Functions may return a type of returnType, but does not need to have any returnType if nothing is returned. See Section 4.1 for more on user-defined functions.

1.2 Game Loop

Although the order of that the blocks appear in the program does not matter, it is suggested that programmer write the blocks in the following order:

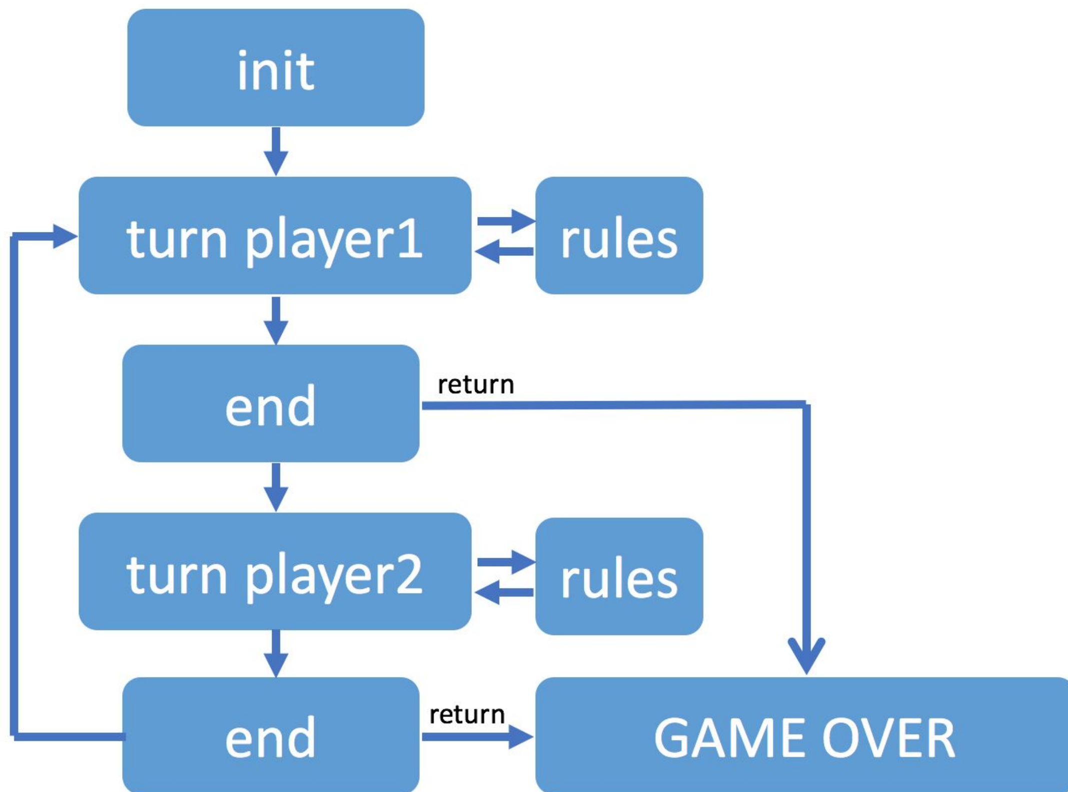
- classes and rules
- init
- functions
- turns
- end

Every game must contain:

- init block
- end block
- at least one turn block

All variables, objects, and game board defined in the init block are globally scoped. In general and for organization, reuseable variables should be initialized in the init block. Otherwise, all variables initialized in the other code blocks have a scope marked within the curly braces.

The following control flow diagram is an example of a game with two players. The init block is executed first to set up the game. Following the init block, there is an internal game loop that continuously iterates through all turn blocks in the game, checking the end block condition after every turn. The game is over when the end returns indicating end of game. Rules blocks may be called anytime throughout the game, usually in the turn blocks.



2 Lexical Elements

2.1 Identifiers

Identifiers are strings for identifying variables and functions. They can contain letters, numbers, and underscores. Identifiers always begin with a letter and are case sensitive, generally beginning with lowercase letters, unless they are a class name.

2.2 Reserved Keywords

Keywords are reserved for specific usages in the language, begin with letters, and cannot be used as identifiers. Keywords are case sensitive.

tiler's keywords include:

```
init, turn, end, rules, class, function
int, string, bool, null
if, else, do, while, for
grid, background, print, prompt, capture, random
True, False
attr, return
```

2.3 Literals

2.3.1 String literals

String literals are sequences of zero or more characters enclosed in single or double quotes. If a quotation character is desired in the literal, either the non-desire quotation character must be used to contain the literal or the quotation character within the literal must be escaped. Escape characters include:

`\n` for new line
`\t` for tab
`\r` for carriage return
`\\` for backslash
`\"` for double quotes
`\'` for single quotes

2.3.2 Integer literals

Integer literals are optionally-signed sequences of digits (0-9), which represent whole numbers and are in decimal format. The sign of the integer literal can be indicated using the '+' or '-' character.

2.3.3 Boolean literals

Boolean literals can take on the values of `True` or `False`.

2.4 Delimiters

2.4.1 Parentheses

Used to enclose tuples, arguments to function calls, expressions within control flow statements, and for defining explicit order of operations within arithmetic and logical expressions.

2.4.2 Commas

Used to separate elements within tuples and arrays, to separate arguments within function calls

2.4.3 Square brackets

Used for array initialization, assignment, and access. Also used for attribute declaration.

2.4.4 Curly brackets

Used to define scope of blocks of code. Can only be used for the predefined code blocks.

2.4.5 Semicolons

Used to indicate end of statement

2.4.6 Period

Used to access attributes and operations of Objects

2.5 Whitespace

Whitespace is used to separate tokens, but is otherwise ignored by the compiler and used for programmer readability only.

2.6 Comments

Comments can be made using the single-line notation (“`//`”) or multi-line comments can be enclosed within (“`/* */`”)

2.7 Operators

`+ - * / %`	add, subtract, multiply, divide, modulo
`== !=`	equal, not equal
`&& || !`	and, or, not
`> < >= <=`	inequality operators
`=`	assignment

2.8 Control Flow

2.8.1 Conditional statement

Conditionally runs code inside curly braces denoted by these statements

```
if (condition) { … } else { ... }
if (condition) { … } else if (condition) { … }
```

2.8.2 `while` loop

Runs code inside curly braces if condition specified by the `while` loop is true

```
while (condition) { … }
```

2.8.3 `do while` loop

Alternative form of the while loop where the actions are written first followed by the condition

```
do { actions } while (condition)
```

2.8.4 `for` loop

Parameters for iteration are start index, end condition, and a increment/decrement operation to the index

```
for (int i = 0; i < end; i++) { … }
```

2.8.5 `for each` loops

Parameters for iteration are common object iterated on and an iterable (array, etc)

```
for item in array { … }
```

3 Datatypes

3.1 Primitive Types

3.1.1 null

Returned any time a value should be returned but none exists. Examples of when `null` may be returned include:

- when the grid is accessed, using grid[x,y], at a position in which no object occupies
- when the coord attribute is accessed, using obj.coord, on an object which is not occupying the grid
- when an object is declared but not instantiated, and its attributes are accessed

3.1.2 int

Can take on the values of integers. Operations include:

```
+, -, *, /, %, ==, !=, <, >, <=, >=
```

3.1.3 float

Can take on values of floating point numbers with a mantissa and exponent. Operations include:

```
+, -, *, /, ==, !=, <, >, <=, >=
```

3.1.4 string

Can take on the value of any sequence of one or more characters. String must always written between single or double quotes. Operations include:

`==, !=, +` (concatenation)

3.1.5 bool

Can take on the value of True or False. Operations include:

```
==, !=, !, &&, ||
```

3.1.6 tuple

A pair of integers, typically representing a coordinate on the grid. Operations include:

```
==, !=, +, -
```

3.2 Non-primitive Types

3.2.1 Array

A one-dimensional list of primitive values.

- Operations:
    - iteration in "for each" construct
    - access with `[]` (eg: `a[5]`)

3.2.2 Object

A collection of attributes of primitive types. Defined in class block.

- Operations
    - obj.[attr_name] (accessing attribute value)
    - = (assignment)
    - obj.move(tuple) (change coordinates of the object while checking that rules are satisfied)
    - obj.isNull (checks if current object is null)
    - obj.delete (remove from grid)

3.2.3 Grid

A 2D array of objects.

- Operations:
    - grid[tuple] (get object at position)
    - grid.row[int] (returns an array of objects whose first index is the same as the int)
    - grid.column[int] (returns an array of objects whose second index is the same as the int)
- Can iterate over the grid slots in a "for each" construct, just like with an array

3.3 Variable Declarations

3.3.1 Local declarations

Local declarations are made in code blocks besides `init`. Consists of two elements: type name and identifier. The type name is either the reserved word for a primitive type (int, string, bool, tuple), or it is a class name. The scope is the block in which they are declared. Done using reserved keyword for primitive type (int, string, bool, tuple)

eg: `int kingCount`

3.3.2 init declarations

Declarations made in the init code block look the same as local declarations. However, if variables are declared here then they are available in all other code blocks.

3.3.3 Attribute declarations

Attribute declarations define attributes of classes using "attr" reserved word. Programmer specifies name and type of variable. Optionally programmer can also specify which values the variable can take on, in a comma-separated list enclosed by square brackets. Syntax: attr [type_name]: [type] [[values]]. For floats, instead of enumerating values, a range is given.

eg: `attr lives: int [0,1,2,3]`
eg: `attr score: int`
eg: `attr health: float [0,100]`

3.3.4 Non-primitive type declarations

- arrays - are declared using array keyword, followed by type and identifier. Type can only be primitive types. Programmer must also list array elements in square brackets. Syntax: array [type] [name] = [[elements]]
  - eg: `array int a = [2,3,5,7,11,13];`

- objects - declared using class name and identifier. Syntax: [Class_Name] [identifier] Can be done through local or init declaration but not attribute declaration.
  - eg: `Player p1`
- grid - must be declared only once in the program, and declaration must be in the init block. Uses keyword "grid" and a tuple representing the number of pieces in each row and column
  - eg: `grid(4,4);`

3.4 Classes

3.4.1 Default Attributes

Each class by default has these attributes. Default attributes are only modifiable by the language's built-in functions, not through assignment. However, they can be accessed by the programmer and operated on according to their types.

- coord: tuple which stores the coordinates of the object on the grid
- x,y: two integers which also store the coordinates of the object on the grid where x is the row and y is the column.

- image: a string which stores either the file path to an image or a string containing "c_[name]" where [name] is a predefined color name.
- className - a string containing the name of the class of which the object is an instance

3.4.2 Instantiation

Instantiation uses the "new" keyword. Programmer lists class name, then in parentheses assigns the values of each of the attributes defined in that class, formed as a comma-separated list. Expression value is an object with those attribute values. Syntax: new [Class_Name]([attr1] = [value1], [attr2] = [value2] ...)

eg: `new Player(health=100, speed=6)`

3.4.3 Object class type

Programmer can assign an object of any type to a variable of type Object

3.4.4 Assignment

Use the '=' operator for assignment. Syntax: [LHS] = [RHS]. Can only assign an object to an expression which has the same type, or type Object.

4 Functions

4.1 User-Defined Functions

User can define a function, functionName, that takes a list of parameters and returns a type of returnType. If functionName does not have a return type, returnType should be set to void.

```
function returnType functionName([parameters]) {
   // developer code here
}
```

To call the function, use the functionName followed by a list of parameters in parentheses.

```
functionName([parameters]);
```

4.2 Built-In Functions

4.2.1 Board Specific Functions

- `grid(x,y)`
  Generates a board defined by a grid of x tiles by y tiles

- `background("filePath")`
  Sets the background of the board to an image specified by the file path, filePath

4.2.2 Input-Output Specific Functions

- `capture()`
  Returns the coordinate of the tile that is selected by the mouse cursor

- `print(messageString)`
  Prints out message_string to output

- `prompt(messageString)`
  Prints out messageString to output and returns an input from the user

4.3 Random Generator Functions

- `random()`
  Returns a random float between 0 and 1

- `random(int n)`
  Returns a random int between 0 and n

- `random(grid)`
  Returns a random coordinate on the grid

4.4 Object Specific Functions

- `object.move(coordinate)`
  Moves an object from its current position to given the position defined by coordinate. Coordinate should be a tuple of the form (x, y) where x is the column and y is the row values of the grid.

- `object.delete()`
  Destroys the object specified and removes it from the grid

- `object.isNull()`
  Returns true or false depending on whether or not the specified object is a Null object.

5 Sample Code

```
// tic tac toe

class Piece {
   attr player: string ['x', 'o'];
}

init {
   grid(3, 3);
   background('3x3grid.jpg');
   int x, y;
}

function playTurn(string playerName) {
   print(playerName + ' turn: click an empty tile');

   // keep capturing mouse coordinates until it captures a valid (empty) tile
   do {
      (x, y) = capture();
   } while (!grid[x, y]);
   grid[x, y] = new Piece(player=playerName);
}

turn {
   playTurn('x')
}

turn {
   playTurn('o')
}

end {
   for piece in grid.row[0] {
      if (piece.player == grid[0,0].player) {
         return grid[0,0].player;
      }
   }

   for piece in grid.row[1] {
      if (piece.player == grid[0,1].player) {
         return grid[0,1].player;
      }
   }
```

```
    for piece in grid.row[2] {
        if (piece.player == grid[0,2].player) {
            return grid[0,2].player;
        }
    }

    for piece in grid.column[0] {
        if (piece.player == grid[0,0].player) {
            return grid[0,0].player;
        }
    }

    for piece in grid.column[1] {
        if (piece.player == grid[1,0].player) {
            return grid[1,0].player;
        }
    }

    for piece in grid.row[2] {
        if (piece.player == grid[2,0].player) {
            return grid[2,0].player;
        }
    }

    if (grid[0,0].player == grid[1,1].player &&
            grid[0,0].player == grid[2,2].player) {
        return grid[0,0].player;
    }

    if (grid[0,2].player == grid[1,1].player &&
            grid[2,0].player == grid[1,1].player) {
        return grid[1,1].player;
    }

    bool gameFull = True;
    for piece in grid {
        if (piece.isEmpty) {
            gameFull = False;
        }
    }

    if (gameFull) {
        return "DRAW";
    }
}
```

```
// chess

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

rules Piece {
   player: 'white' && role: 'pawn' >> [(x, y+1)];
   player: 'black' && role: 'pawn' >> [(x, y-1)];
   player: 'knight' >> [(x+1, y+2), (x-1, y-2), (x-1, y+2), (x+1, y-2),
                        (x+2, y+1), (x-2, y-1), (x+2, y-1), (x-2, y+1)];
      .
      .
      .
}

init {
   grid(8, 8);
   background('chess.jpg');
   int x, int y;

   for piece in grid.row[1] {
      piece = new Piece(player='black', role='pawn', removed=False);
   }
   grid[0,0] = new Piece(player='black', role='rook', removed=False));
   grid[0,7] = new Piece(player='black', role='rook', removed=False);
   grid[0,1] = new Piece(player='black', role='knight', removed=False);
   grid[0,6] = new Piece(player='black', role='knight', removed=False);
   grid[0,2] = new Piece(player='black', role='bishop', removed=False);
   grid[0,5] = new Piece(player='black', role='bishop', removed=False);
   grid[0,3] = new Piece(player='black', role='queen', removed=False);
   grid[0,4] = new Piece(player='black', role='king', removed=False);

   for piece in grid.row[6] {
      piece = new Piece(player='white', role='pawn', removed=False)
   }
   grid[7,0] = new Piece(player='white', role='rook', removed=False);
   grid[7,7] = new Piece(player='white', role='rook', removed=False);
   grid[7,1] = new Piece(player='white', role='knight', removed=False);
   grid[7,6] = new Piece(player='white', role='knight', removed=False);
   grid[7,2] = new Piece(player='white', role='bishop', removed=False);
   grid[7,5] = new Piece(player='white', role='bishop', removed=False);
   grid[7,3] = new Piece(player='white', role='queen', removed=False);
   grid[7,4] = new Piece(player='white', role='king', removed=False);

}
```

```
function playTurn(string playerName) {
   print(playerName + ' turn: select a piece');

   do {
      (x, y) = capture();
   } while(grid[x,y].player == playerName);

   Piece piece = grid[x,y];

   print('Select a tile to move to');
   do {
      (x, y) = capture();
   } while(piece.rules(x, y));

   if (grid[x,y]) {
      grid[x,y].delete();
   }
   piece.move(x, y);
}

turn {
   playTurn('white');
}

turn {
   playTurn('black');
}

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

```
// The Wumpus World - modified

class Character {
   attr lives: int [0, 1, 2, 3];
   attr level: int [1, 2, 3, 4, 5];
   attr goldPieces: int;
}

rules Character {
   level: 1 || level: 2 || level: 3 >> [(x+1, y), (x-1, y),
                                         (x, y+1), (x, y-1)];
   level: 4 || level: 5 >> [(x+1, y+1), (x-1, y-1), (x-1, y+1), (x+1, y-1)];
}

class Obstacle {
   attr type: string ['pit', 'breeze', 'stench', 'wumpus'];
}

init {
   grid(4, 4);
   background('4x4grid.jpg');
   int x;
   int y;

   Character me = new Character(3, 1, 0);
   me.move(0, 0);

   int pitCount = 0;

   while (pitCount < 3) {
      (x, y) = random(grid);

      // check if there is an object on grid at location (x, y)
      if (!grid[x,y]) {
         Obstacle pit = new Obstacle('pit');
         pit.move(random(grid));
         pitCount = pitCount + 1;
      }
   }

   ...

}
```