GridLok

Language Reference Manual

Alice Hwang {ah2813}
Bryan Yu {by2181}
Julian Edwards {jse2122}
Laura Hu {lh2718}

Table of Contents

1 Overview

1.1 Goals

The purpose of Gridlok is to more easily facilitate the creation of grid-based games, such as Tic-Tac-Toe, Minesweeper, or even Chess. Rather than having excessively complicated for loops to check the values in a column or to check certain grid coordinates, Gridlok has simplified functions to do so such as for piece in col(colNum). Additionally, Gridlok has language specific functions for the grid such as checking the 8 grid locations around a piece, check a grid location relative to a piece, etc.

2 Lexical Elements

2.1 Identifiers

Identifiers are used to name variables and functions. They are case sensitive. Only letters and digits can be used in identifiers. The first character of an identifier cannot be a digit and must be a lowercase character.

2.2 Keywords

Keywords are identifiers that are reserved for the programming language.

The following are a list of keywords used by GridLok:

game	board	dimensions	image	
players	turnOrder	piece	name	
onClick	visible	return	def	
fun	if	elseif	else	
for	setup	place	in	
row	col	surrounding	AND	
OR	winCondition	loseCondition		drawCondition
all	display	print	onTurn	
eventPriority				

2.3 Constants

Built in constants that also may not be used for identifier names include:

true	false	empty

2.4 Pre-defined Variables

Predefined variables that may not be used for identifier names include:

rows	cols	win	lose
draw	turnCount	pieceRow	pieceCol
turn			

2.5 Operators

Operators are tokens used to perform actions, such as addition or subtraction, on operands. More will be discussed in section 4.

2.6 Separators

Separators separate tokens. White space is a separator, but not a token. Other separators are themselves single character tokens.

Separators in our language include:

```
{}()[],
```

2.7 White Space

White space is represented by a set of characters: the space character, the tab character, and the newline character. Whitespace is ignored, except when it is used as a separator for tokens or within a string constant.

```
piece{name{“my piece”}}
```

and

```
piece{
      name{“my piece”}
}
```

are equivalent.

2.8 Comments

GridLok only supports multiple lined comments. Comments must begin with /* and end with */

3 Data Types

3.1 Primitive Data Types

The primitive data types in GridLok reflect the simplicity and purpose of the language. There are only three primitive data types that are more than sufficient for coding grid-based games.

3.1.1 Integers

The int data type will represent integer values between −2,147,483,648 to 2,147,483,647. It consists of a sequence of digits. Our language does not support decimal values.

3.1.2 Strings

Any text will be of type string. For simplicity, there will be no character data type, just one letter strings. A string a sequence of zero or more characters and digits enclosed within double quotes.

3.3.3 Booleans

Booleans can either be true or false, and are manipulated by the !, AND, and OR operators.

3.2 Non-Primitive Data Types

The non-primitive data types, like their primitive counterparts, maintain both the simplicity and purpose of GridLok.

3.2.1 Lists

Users can create variable-sized lists where elements can be appended to the end of the list only. All elements in a list must be of the same data type.

3.2.1.1 Declaring Lists

Lists are declared using the def keyword as follows. Note that an empty list must include a comma and each element, including the last, in the list must have a comma following it.

```
def myList{element1, element2,..., element k,}

Example: def myList {,}
Example: def myList {3,}
Example: def myList {3,2,5,}
```

3.2.1.2 List Functions

Lists have simple functions for accessing, appending, changing, and removing elements.

Access an element, returns element:
```
myList[index]
Example: myList[5]
```

Append an element, appends to end of list:
```
myList {element}
Example: myList {“world”}
```

Change an element:
```
myList[index] {new_element}
Example: myList[3] {“bye”}
```

3.2.2 Piece

The piece data type is used for pieces that are placed in a board during a game.

3.2.2.1 Declaring a Piece

Pieces are declared with the name of the piece as a string, the pathfile for the image as a string, and the onClick specification.

```
piece {
      name {string piece_name}
      image {string pathfile}
      onTurn {
            /* optional triggers for specific events */
      }
      onClick {
            /* events that must occur when piece is clicked go here */
            /* can be left blank */
      }
}
```

Example:
Declaring a piece called “mines” for a Minesweeper game.
```
piece {
      name {“mines”}
      image {“mine.png”}
      onTurn{}
```

```
	onClick {
		visible {1} /* sets piece to visible on board */
		endTurn {}
	}
}
```

3.2.3 Board

The board data type is used for creating a board for the game with specific dimensions. Exactly one board is required and allowed to be declared at the beginning of each program.

3.2.3.1 Declaring a Board

All that is required for declaring a board is the dimensions which is how many rows by columns the user wants and the pathfile for the image for the board.

```
board {
	dimensions {int numRows, int numCols}
	image {string pathfile}
}
```

Example:
Declaring a board for a tic-tac-toe game.

```
board {
	dimensions {3,3}
	image {“ttt.png”}
}
```

4 Expressions and Operators

4.1 Expressions

An expression consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values.

Examples include:

```
	2 + 2
	rows + cols
	rows + 1
	win OR lose
```

4.2 Assignment Operators

Assignment operators store values in variables. In order to assign a value to a variable, you must declare a variable using the “def” keyword and specify the variable name and value. The general form is as follows:

```
def variableName {variableValue}
```

While variableValue can be a constant, expression, or another variable. If passing in another variable as the value, GridLok will copy the value, not the reference.

Strings must be specified with double quotation marks, and lists must be specified using commas.

4.3 Arithmetic, Comparison, and Logical Operators

The table below shows the arithmetic, comparison, and logical operators supported by the GridLok language, listed in order of precedence from highest to lowest.

Operator	Category	Description	Associativity
!	Logical	Logical Not	Right-to-left
* / %	Arithmetic	Multiplication, integer division, modulo	Left-to-right
+ -	Arithmetic	Addition, subtraction	
< <= > >=	Comparison	Less than, less than or equal to, greater than, greater than or equal to	
== !=	Comparison	Equal, not equal	
AND	Logical	Logical And	
OR	Logical	Logical Or	

5 Statements

5.1 if Statement

The if statement is used to execute part of a code based on the truth value of an expression.

General format:

```
if(test){
      then-statement
}
else{
      else-statement
}
```

Series of if statements:

```
if(test){
      then-statement
}
elseif(test){
      then-statement
}
elseif(test){
      then-statement
}
else{
      else-statement
}
```

5.2 for Loops

5.2.1 Enhanced for Loops

`for variableName surrounding int x, int y`
  Loops through all the pieces directly adjacent to coordinates (x,y).

`for variableName in row(int rowNum)`
  Loops through all pieces in the row corresponding to rowNum.

`for variableName in col(int colNum)`
  Loops through all pieces in the column corresponding to colNum.

`for int i, int j, surrounding int x, int y`
  Loops through all coordinates i,j directly adjacent to coordinates (x,y).

5.2.2 Looping over integers

`for variableName(int min, int max)`
  Loops through min (inclusive), max (exclusive).

`for variableName(int min, int max), variableName2(int min2, int max2)`
  A nested for loop with the second loop being the nested loop.

`for int x, int y in board`
  Loops through all the locations in the board.

6 Functions

6.1 Function Declarations and Definitions

Function declarations and definitions are used to specify the name of a function and a list of parameters. The reserve word “fun” is used to begin a function declaration, and “return” is used in the function body to specify the return value, if any. A general form is:

```
fun function-name(parameter1, parameter2 … ){ function-body }
```

Note that you cannot have a function and a variable of the same name within the same scope and that Gridlok does not support overloading functions.

6.2 Calling Functions

Functions are called using their names with the parameters between curly braces. For example, calling a function called add with the function definition:

```
    fun add(a, b) {return a + b}
```

can be called like so:

```
    add{3,5}
```

6.3 Built-in Functions

```
click {int x, int y}
```

Clicks on a given coordinate (x,y) on the board. If a piece is present, then the its onClick method will be called.

```
removePiece {int x,int y}
```
Calling this removes this piece at the specified coordinates (x,y) from the board. If the coordinate is empty, removePiece does nothing.

```
removePlayer {string playerName}
```
Removes the player. Note that the pieces belonging to the player will remain on the board unless removed.

```
changeType {string pieceName}
```
A function specific to piece that changes the piece type to the name specified.

```
endTurn {}
```
Ends turn so that the program can go onto the next player’s turn after checking win/loss/draw condition.

```
rand {int x, int y}
```
Returns a random integer between x (inclusive) and y (exclusive).

```
print {x}
```
This function prints x to console, it accepts any data type. Under the hood x is converted to a string before it is printed.

```
display {x}
```
Displays the x to the board, it accepts any data type. Under the hood x is converted to a string it is displayed.

7 Event Handling

7.1 User Defined Events

Event handling in GridLok refers to the execution of a function based on global booleans that are triggered upon a certain predefined condition. Multiple events can be ordered by priority in a list defined by the user.

Outside of piece definition:
```
eventPriority {string pieceName, string pieceName}
```

Must be inside piece definition:
```
onTurn {
      /* triggers and events */
      /* can be left blank */
}
```

8 Scope

8.1 Scope

Scope refers to what parts of the program can “see” a declared object. A declared object can be visible only within a particular function. Declarations made at the top-level of a program (i.e., not within a function) are visible to the entire program, including from within functions, but are not visible outside of the

program. Declarations made within functions are visible only within those functions.
A declaration is not visible to declarations that came before it; for example:

```
    def x {5}
    def y {x + 10}
```

will work, but:

```
    def x {y + 10}
    def y {5}
```

will not.

9 Sample Programs

9.1 minesweeper.gl

```
game{
        board {
                dimensions {10, 10}
                image {board.png}
        }

        players {“player”}
        turnOrder {“player”}

        piece {
                name {“mines”}
                image {“m.png”}
                onTurn {}
                onClick {
                        visible{true}
                        endTurn{}
                }
        }

        piece {
                name {“zero”}
                image {“zero.png”}
                onTurn{}
                onClick {
                        visible{true}
                        for i,j surrounding pieceRow, pieceCol {
                                if board[i][j].name!=“mine”{
                                        click{i,j}
                                }
                        }
                        endTurn{}
                }
        }

        piece {
                name {“one”}
                image {“one.png”}
                onTurn{}
                onClick {
                        visible{true}
                        endTurn{}
                }
        }

        /*do this for 2-8*/

        setup{ //set turnCount=0
                player {
                        for i(0,10){
                                def x(rand{0,rows}}
                                def y{rand{0,cols}}
                                if board[x][y] == empty{
                                        place{“mines”,x,y}
                                }
                                else{
                                        i{i - 1}
                                }
                        }

                        for x,y in board{
                                def count {0}
                                if board[x][y]==empty{
                                        for p surrounding x,y{
                                                if p.name == “mines” {
                                                        count {count + 1}
                                                }
                                        }
                                        if count == 0 {
                                                place {“zero”, x,y}
                                        }
                                        if count == 1 {
                                                place {“one”, x,y}
                                        }
                                        if count == 2 {
                                                place {“two”, x,y}
                                        }
```

```
                    if count == 3 {
                        place {“three”, x,y}
                    }
                    if count == 4 {
                        place {“four”, x,y}
                    }
                    if count == 5 {
                        place {“five”, x,y}
                    }
                    if count == 6 {
                        place {“six”, x,y}
                    }
                    if count == 7 {
                        place {“seven”,x,y}
                    }
                    if count == 8 {
                        place {“eight”, x,y}
                    }
                }
            }
        }
    }

    winCondition{
        def temp{true}
        for all sp in board{
            temp{temp AND sp.name!= “mine” AND sp.visible}
        }
        win{temp}

    }

    loseCondition{//there is a visible mine
        def temp{false}
        for all sp in board {
            temp{temp OR (sp.name == “mine” AND sp.visible)}
        }
        lose{temp}
    }
}
```

9.2 tic-tac-toe.gl

```
game{
    board{
        dimensions {3,3}
        image {ttt.png}
    }
```

```
players{“x”, “o”}
turnOrder{“x”, “o”}

piece{
        name {“placeHolder”}
        image{“ph.png”}
        onTurn{}
        onClick{
                if turn==”x”{
                        changeType{“xPiece”}
                }
                else{
                        changeType{“oPiece”}
                }
                endTurn{}
        }
}
piece{
        name {“xPiece”}
        image{“x.png”}
        onTurn{}
        onClick{}
}
piece{
        name {“oPiece”}
        image{“o.png”}
        onTurn{}
        onClick{}

}

setup{
        for x,y in board{
                place{“placeHolder”,x,y}
        }
}

winCondition{
        /* Checks the rows for matching pieces */
        for x(0,rows){
                def tempX {true}
                def tempO {true}
                for p in row(x){
                        tempX{tempX AND p.name==”x”}
                        tempO{tempO AND p.name==”o”}
                }
                win{win||tempX||tempO}
```

```
            }
            /* Checks the columns for matching pieces */
            for y(0,cols){
                    def tempX {true}
                    def tempO {true}
                    for p in col(y){
                            tempX{tempX AND p.name==”x”}
                            tempO{tempO AND p.name==”o”}
                    }
                    win{win||tempX||tempO}
            }
            /* Checks the diagonals for matching pieces*/
            if((board[0][0] == board[1][1] AND board[0][0] == board[2][2]) OR (board[0][2] ==
            board[1][1] AND board[0][2] == board[2][0])) {
                    win{true}
            }
    }
    drawCondition{
            def instancesOf{0}
            for x,y in board{
                    if board[x][y].name==”placeHolder”{
                            instancesOf{instancesOf + 1}
                    }
            }
            draw{instancesOf==0 && !win}
    }

}
```