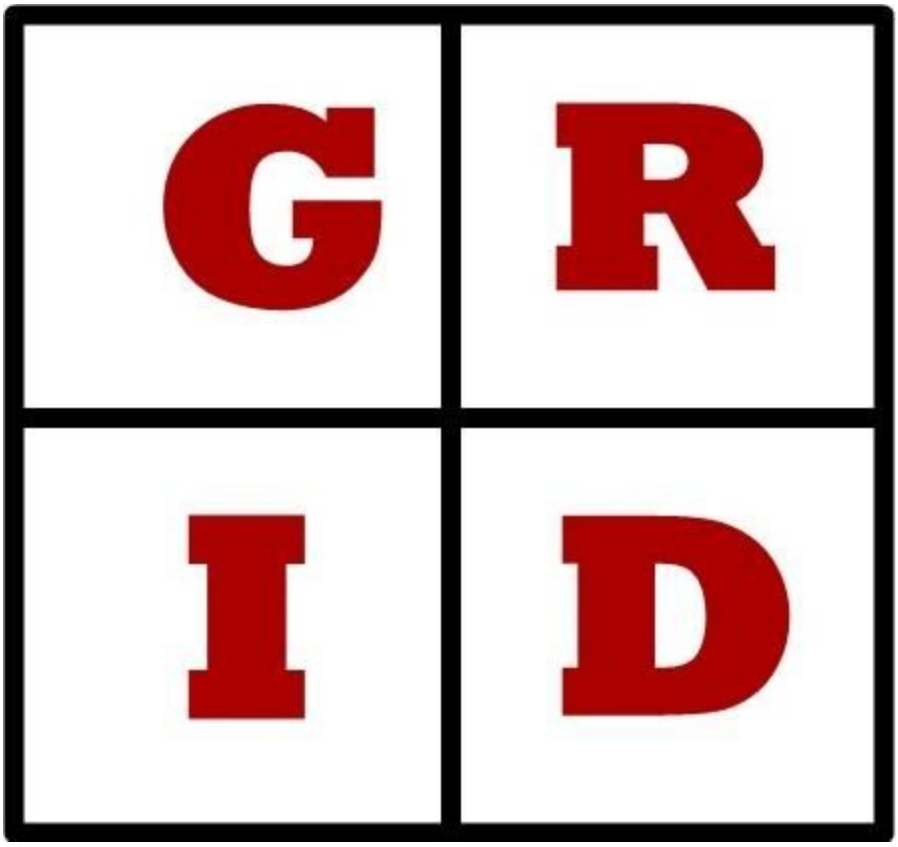


GridLang: Grid Based Game Development Language

Language Reference Manual

Programming Language and Translators - Spring 2017
Prof. Stephen Edwards

Akshay Nagpal	**an2756**@columbia.edu
Dhruv Shekhawat	**ds3512**@columbia.edu
Parth Panchmatia	**psp2137**@columbia.edu
Sagar Damani	**ssd2144**@columbia.edu

Table of Contents

1. Lexical Elements

Identifiers are strings for naming variables and functions. The identifiers are case-sensitive. They can consist of letters, digits and underscores, but must always start with a letter.

```
X      x      a1     b2
```

1.2 Keywords

Keywords are reserved for use in the language and cannot be used as identifiers. These keywords are case sensitive.

int	function	player	item	grid
coordinate	float	colocation	gameLoop	while
checkMove	return	string	drawGrid	rand
gameOver	createGrid	traverse	boolean	checkGameEnd
type	repeat	print	prompt	playerOrder
p				

1.3 Literals

Literals are constant string, numeric, or boolean values, such as “abc”, 25, or false. Each literal is immutable and has a specific data type corresponding to one of the primitive types. No type casting is allowed. Trying to assign a literal to a variable of mismatching type will cause an error.

```
string x = "abc"
```

1.3.1 String literals

String literals are a sequence of zero or more non-double-quote characters and/or escaped characters, enclosed in double quotes. An escaped character is a character that immediately follow a backslash ‘\’. The backslash signals to the compiler to interpret the escaped character in a special way, according to the following table:

```
\n                      Insert a newline
```

```
\t                          Insert a tab
\\                        Insert a backslash
 \"                        Insert a double quote
```

1.3.2 Integer literals

Integer literals are whole numbers represented by a sequence of one or more digits from 0-9. Integers are assumed to be in decimal (base 10) format. Precede an integer with “- ” to denote a negative number.

```
31      -343
```

1.3.3 Boolean literals

A boolean literal represents a truth value and can have the value true or false (case sensitive).

```
true  false
```

1.4 Delimiters

Delimiters are special tokens that separate other tokens, and tell the compiler how to interpret associated tokens.

1.4.1 Parentheses and Braces

Parentheses are used to force evaluation of parts of a program in a specific order. They are also used to enclose arguments for a function.

```
int x = 10, y = 20, z = 30, w = 40
w = (x+y) * z    # here, (x+y) will be executed first and then multiplied with z
```

1.4.2 Commas

Commas are used to separate function arguments and variable declarations.

1.4.3 Square Brackets

Square brackets are used for array initialization, assignment, and access.

```
arrayName[0] = 3
```

1.4.5 Curly Braces

Curly braces are used to enclose function definitions and blocks of code. In general, blocks enclosed within curly braces do not need to be terminated with semicolons.

1.4.6 Periods

Periods are used to access attributes of Item and Player data types, as well as to define the event handlers on these types.

1.4.7 New Line Characters

New Line Characters (either \n or \r) are used to terminate a sequence of code.

1.5 Whitespace

Whitespace (unless used in a string literal) is used to separate tokens, but has no special meaning otherwise. List of whitespace characters: spaces, tabs and newlines.

2.Data Types

2.1 Primitive Data Types

2.1.1 int

These are 32-bit signed integers that can range from –2,147,483,648 to 2,147,483,647

2.1.2 string

All text values will be of this type.

2.1.3 boolean

A truth value that can be either true or false.

2.1.4 coordinate

An ordered pair of non-negative integers, used to index the grid.

```
(2,3)
```

2.1.5 void

Used only as a return type in a function definition; specifying void as the return type of a function means that the function does not return anything.

2.2 Non-primitive Data Types:

2.2.1 Arrays

Arrays are ordered, fixed-size lists that can be used to hold both primitive and non-primitive data-types. All elements of an array must be of the same type. An array must be initialized with its size.

2.2.1.1 Declaring Arrays

You can declare an array by indicating the type of the elements that the array will contain, followed by brackets enclosing the number of elements an array will hold, followed by an identifier for the array. For example:

```
int[5] myArray
```

This declares an array named myArray that can hold 5 integers.

2.2.1.2 Accessing and setting array elements

Array elements can be accessed by providing the desired index of the element in the array you wish to access enclosed within brackets next to the identifier of the array. For example:

```
myArray[1]
```

This returns the element in myArray at index 1.
Array elements can be set by accessing the element via the desired index in which to place the item and then assigning the desired value to the entry. For example:

```
myArray[1]=4
```

sets the element in myArray at index 1 to 4.

2.2.1.3 Array Length

To get an array's length, simply call the len function on the array.
For example:

```
int size=len(my_array)
```

2.2.2 Grid

Grid is a global 2-dimensional array of cells. Each cell holds an array of Players (called playerList) and an array of Items (called itemList). A specific cell in the grid can be referenced by its coordinates. Eg: Grid(3,4) will be the grid cell on the fourth row and the fifth column. These can be referenced as Grid(3,4).playerList and Grid(3,4).itemList.

2.2.3 Player

Player has the following in-built attributes:

- position: a coordinate denoting the position of the Player on the grid.
- win: a boolean value. If this values is set, then the Player is removed from the playerOrder.

- displayString: a string that will be displayed at the position of the Player in drawGrid()
- type: an enum that denotes the type of the Player
- Exists: boolean. When this is set to false, the Player is removed from the gameloop, and removed from the Grid.

Moreover, additional attributes can be added to Player using the following syntax:

```
Player {
	Datatype attr1
	Datatype attr2
	Datatype attr3
	#specify player-type specific attributes
	type = "evil" {
		Datatype attr4
	}
	type = "good" {
		Datatype attr5
	}
}
```

Handlers can be defined on players for colocation and movement. These will be described further in Section 6.

2.2.4 Item

An item is anything on the board other than the player. Item has the following attributes:

- position: a coordinate denoting the position of the Item on the grid.
- displayString: Denotes the string that will be displayed at the position of the Item in drawGrid().
- Exists: boolean. When this is set to false, the item is removed from the Grid.

Moreover, additional attributes can be added to Item using the following syntax:

```
Item ItemName {
	Datatype attr1
	Datatype attr2
}
```

3 Expressions and Operators

3.1 Expressions

Expressions are made up of one or more operands and zero or more operators. Innermost expressions are evaluated first, as determined by grouping into parentheses,

and operator precedence helps determine order of evaluation. Expressions are otherwise evaluated left to right.

3.2 Operators

The table below presents the language operators (including assignment operators, mathematical operators, logical operators, and comparison operators), descriptions, and associativity rules. Operator precedence is highest at the top and lowest at the bottom of the table.

- '+' (Addition) : Adds values on either side of the operator.
- '-' (Subtraction) : Subtracts right-hand operand from left-hand operand.
- * (Multiplication) : Multiplies values on either side of the operator.
- / (Division) : Divides left-hand operand by right-hand operand.
- % (Modulus) : Divides left-hand operand by right-hand operand and returns remainder.
- ++ (Increment) : Increases the value of operand by 1.
- -- (Decrement) : Decreases the value of operand by 1.
- Relational Operators: >, <, =, ==, <=, >=, != (standard meanings)
- Logical Operators: and, or, ! (standard meanings)
- Assignment Operators: =, +=, -=, *=, /=, %= (standard meanings)

4. Statements

4.1 The If Statement

The if statement is used to execute a statement if a specified condition is met. If the specified condition is not met, the statement is skipped over. The general form of an if statement is as follows:

```
if(condition)
{
	statement1
	statement2
	statement3
}
else
{
	statement4
}
```

4.2 The while statement

The while statement is used to execute a block of code continuously in a loop until the specified condition is no longer met. If the condition is not met upon initially reaching the while loop, the code is never executed. The general structure of a while loop is as follows:

```
while(condition)
{
        Action1
        Action2
        Action3
}
```

5.Functions

5.1 Function Definitions

Function definitions consist of an initial keyword “function” a return type, a function identifier, a set of parameters and their types, and then a block of code to execute when that function is called with the specified parameters. An example of an addition function definition is as follows:

```
function sum (int a, int b): int {
        return a+b
}
```

5.2 Calling Functions

A function can be called its identifier followed by its params in parentheses. for example: sum(1,2)

5.3 Built-in functions

5.3.1 drawGrid()

Draws the grid on the console

5.3.2 traverse([Coordinate c1, Coordinate c2], string listType)

Traverses a row, a column or a diagonal in the grid with start coordinate c1 and end coordinate c2. Returns a list of the listType specified. Valid listTypes are “playerList” and “itemList”.
Eg: traverse([(1,1), (3,3)], “playerList”) returns a list of all the players in the cells along the diagonal from (1,1) to (3,3)

5.3.3 random(0,1)

Generate a pseudo-random number between 0.000 to 1.000.

5.3.4 gameOver()

Displays the winner and exits the program.

5.3.5 print(String s)

prints s on console.

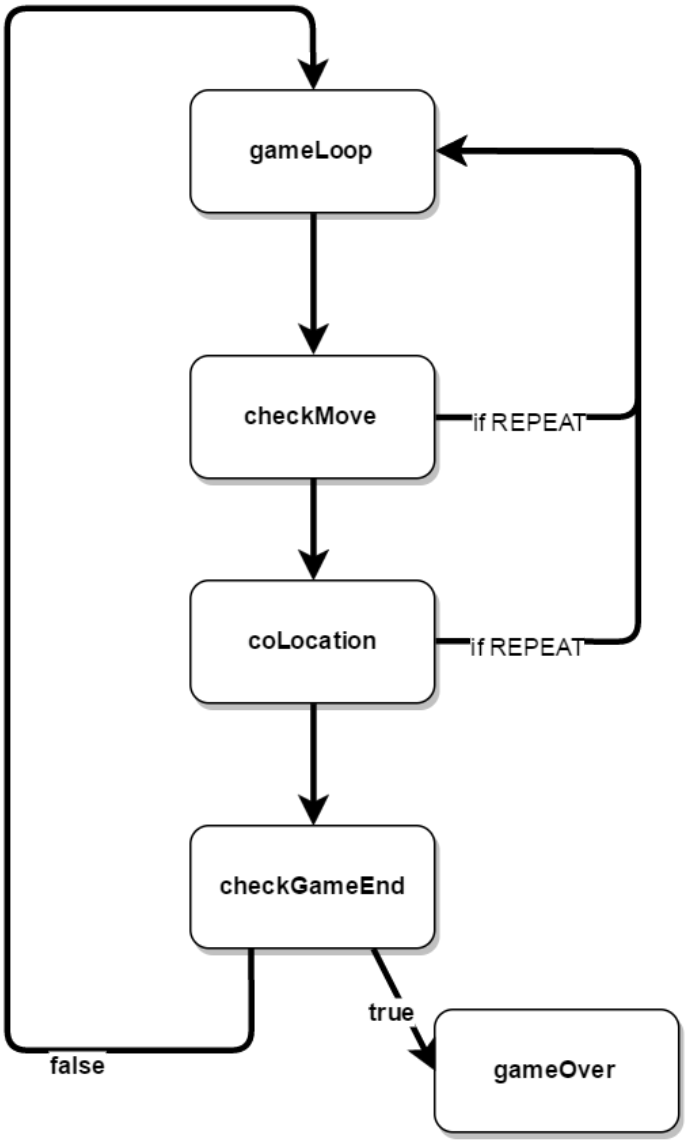
5.3.6 prompt(String msg, dataType)

prompts user for input of a certain dataType.

6. Program Structure and Scope

The entire game code will be in 1 source file.

The flow followed by the code, written by any developer, will be as follows:



The aforementioned blocks imply the following:

6.1 Layout(int rowCount, int colCount)

This block of code is executed first. It is used to set up the game Grid. It takes two parameters: rowCount and colCount, which describe its size. Creation of items, players, playerOrder and the location of these items and players on the Grid will be described here.

6.2 gameLoop()

This block of code gets executed repeatedly until the game ends. Each execution of this block signifies one turn in the game. The gameLoop reads the order of turns from a

built-in list called playerOrder which is defined by the programmer in the layout block. For e.g

```
playerOrder = ['p1', 'p2', 'p3', 'p4']
```

Whenever a player exits the game the programmer deletes this player from the list. As a result, the gameLoop will only run for the remaining players.
Inside the gameLoop, the programmer can use a placeholder ‘p’ which at runtime gets replaced by the player whose turn is executing.

6.3 checkMove()

This is an event handler for which the trigger is only checked after the gameLoop block finishes one iteration. It is used to verify valid movement of the Player or Item. The trigger for this block of code is the change of the location of a Player or Item in the grid during the gameloop(). If checkMove returns false, the location change of the Player/Item is invalidated. The code inside the checkMove block is written by the programmer.
Eg:

```
bishop.checkMove(coordinate start, coordinate end)
{
	#Developer code here
}
```

6.4 Player/Item.colocation(Player/Item, playerType(optional))

This block of code gets triggered when the Player or Item specified in the argument moves to the same location as the Player or Item on which this colocation handler is defined. It is executed only after the move made is validated by the checkMove block. The significance of this block is to determine if the Player or Item’s resultant position on the board leads to an interaction with an existing Player or Item on the board in that position.

Eg:

```
Bonus.colocation(Player)
{
	Player.score++
}
```

6.5 repeat

If repeat is called anywhere in the program, the control shifts back to the beginning of gameLoop.

6.6 checkGameEnd

This block of code gets executed after every iteration of the gameLoop (and after any colocation events have been handled). The developer would place the logic to test the end of the game in this block. If checkGameEnd returns false, execution returns to the next

iteration of gameLoop(). If it returns true, execution flows to the gameOver block. As in gameLoop(), ‘p’ refers to the current player.

Eg:

```
checkGameEnd()
{
        if (p.score == 100)
                return true
        return false
}
```

6.7 gameOver

This block of code gets executed once checkGameEnd returns true. Announcing the winner of the game, and any clean-up code, can be placed in this block.

```
gameOver()
{
        # The programmer enters code here
}
```