Gridworld: Language Reference Manual

Andrew Phan, Kevin Weng, Loren Weng, Zikai Lin
Uni: ap3243, kw2538, lw2504, zl2442

October 26, 2015

Professor Edwards
Programming Languages & Translators
Columbia University

1 Introduction

Creating a game, whether you are an experienced programmer or not, is a daunting task. There are many in-game related functions, actions, and rules to keep track of, which not only removes the fun out of creating a game but may also scare the user away from creating a game altogether. Gridworld language takes the average user into consideration by focusing on supplying the necessary tools to make game creation a more enjoyable experience. Our goal is to allow the designer to quickly delve into writing Gridworld code, and create a modular game based on the content creator's own storyline, plot, combat system and their own integrated functions.

1.1 Gridworld's Focus

- Simplifies game development, requiring little, if any, programming experience.
- Allows the user to design a game world based on their own rules, needs and specifications.
- Offers useful game-related tools and functions for content creator.

1.2 Language Advantages

- Rapid object/class declarations.
- Easy to add and modify attributes.
- Allows for the randomization of objects in the game world.
- Generates an interactive and customizable 2d board-like map.
- Turn-based game loop focuses on user interaction via a dedicated console.
- Preserves game state and allows user to undo game actions.

2 Lexical Conventions

2.1 Tokens

2.1.1 Keywords

Keywords are pre-defined words in the Gridworld compiler. Each keyword has has their own function. The keywords are reserved in the compiler and therefore cannot be used as variable names. Below is a table of reserved keywords in Gridworld:

Keywords	
if	else
while	for
bool	int
str	print
break	continue
and	or
not	new
return	elif
gameloop	

2.1.2 Identifiers

An identifier is an element in the Gridworld language that has been given a name for a particular function, and variables. An identifier begins with a letter or underscore followed by any alphanumeric characters. Special characters such as punctuation and brackets are not allowed, except for the underscore. Identifiers should give a clear indication of what the label does, if it is an action then a name should define that action. Finally, it should not be a keyword.

```
string player1_Name // player1_Name is the identifier
int hitPoints // hitPoints is the identifier
int sum     // sum is the identifier
bool attackPlayer()     // attackPlayer is the identifier
```

2.2 Whitespaces

Whitespaces is any character or a series of whitespace characters that are unused or space between objects. Their purpose is to separate tokens and format programs. Whitespace characters are usually typed in by using the return, spacebar or the tab key. Gridworld ignores all whitespaces, as the language uses the spacing for differentiating tokens. If we look above at code listing **??** above, int and sum are separated by a whitespace. Unlike Ocaml, Gridworld does not care about indentations.

2.3 Comments

Like most programming languages, Gridworld supports single line and also multi-line commenting. The language encourages readable, organized and logical source code. With that in mind, being able to comment source code not only helps debug Gridworld, but also helps clarify what a function does, why the creator decided to do something, or to create notes and reminders on what the next step is in terms of game development.

Symbol Syntax	Uses	Example
%| |%	Multi-line comments, nested comments	%| This is a multi-line comment, intended to allow the user to provide more info on what's happening during execution |%
%	Single-line comment	% This is a single-line comment.

3 Types/Meaning of Identifiers

3.1 Primitive Types

The Gridworld languages uses several primitive types in order to describe the user's game environment. The primitive types are listed below and may be used, although not limited to, for example describing a world with hit-points, weapon damage, name, and dialog of players, monsters and other objects.

Primitive Type	Description	Range	Example
Int	A 32 bit Integer type	-2.147.483.648 to 2.147.483.647	int maxLife = 2000;
Bool	A Boolean value	True (Binary 1) / False (Binary 0)	bool playerGodlike = TRUE;
String	A sequence of characters	Not Applicable	String bossName = "Diablo"

3.2 Compound Types

3.2.1 Array

An array is a data structure, which is able to store a collection of elements of the same type in consecutive memory. The lowest address in memory corresponds to the first element and the highest address with the last element. All arrays in Gridworld start with an index of 0 for the first element and the last element in the array being the total size minus 1. The arrays in Gridworld can be implemented as 1-dimensional or 2-dimensional arrays.

```
String[] inventoryItems = [sword, shield, boots]; // creates 1d array of 3
    elements
int[][] sizeofMap = new int[100][100];  // creates 2d array of 100x100 int
    elements for xy coordinate system
```

3.2.2 <object>Array

This is almost identical to the array above, however, the difference is the ability to store objects rather than a collection of the same data types.

```
GameObject [] monsters = new GameObject[200]; // creates an array of 200
    monsters
```

4 Expressions and Operators

The precedence of expression operators are indicated by the order of the following subsections, from highest precedence to the lowest.

4.1 Primary Expressions

Identifiers: An identifier refers to a variable or a function.

Constants: A constant can be a number, string, boolean etc. with the different types defined in lexical conventions.

String literals: String literals are directly translated to strings by the compiler.

Parenthesized expressions: The expression is equivalent to the result without parentheses, but the presence of parentheses indicates the precedence as a primary expression.

4.2 Unary Operators

4.2.1 Logical Negation

Types used with the logical negation operator are Bool and Int. The result of the logical negation of a Bool is true if the value of the expression is false, and false if value is true. The result of the logical negation of an Int is 1 if the value of the expression is 0, and 0 if the value of the expression is non-zero.
!expression

4.2.2 Increment Operator

Type used with the increment operator is Int. The value of the expression is incremented by one. The result is the value of the expression incremented by one.
Increment expression++

4.2.3 Decrement Operator

Type used with the decrement operator is Int. The value of the expression is decremented by one. The result is the value of the expression decremented by one.
Decrement expression--

4.3 Function Call

To call a function, it must have been declared and defined previously. A function call has the form functionName(expression1, expression2,...), following the form defined in its declaration. The result is a value of the type defined as a return type in the function declaration.

4.4 Multiplicative Operators

The multiplicative operators are left associative. Types of both expressions used with the * operator are Int. The result is the first expression multiplied by the second.
expression $*$ ***expression***;

Types of both expressions used with the / operator are Int. The result is the first expression divided by the second, and division by zero is not allowed.
expression / expression;

Types of both expressions used with the % operator are Int. The result is the remainder from the division of the first expression by the second. Division by zero is not allowed.
expression % expression;

4.5 Additive Operators

The Additive operators are left associative. Types of both expressions used with the + operator are Int.

The result is the sum of the two expressions.
$\boldsymbol{expression + expression}$;

Types of both expressions used with the - operator are Int. The result is the first expression minus the second.
$\boldsymbol{expression - expression}$;

4.6 Relational Operators

The relational operators are left associative. The type of the relational operators are Int.

The result is of type Bool and the value is true if the first expression is less than the second expression, and false otherwise.
expression < expression;

The result is of type Bool and the value is true if the first expression is greater than the second expression, and false otherwise.
expression > expression;

The result is of type Bool and the value is true if the first expression is less than or equal to the second expression, and false otherwise.
$\boldsymbol{expression <= expression}$;

The result is of type Bool and the value is true if the first expression is greater than or equal to the second expression, and false otherwise.
$\boldsymbol{expression >= expression}$;

4.7 Equality Operators

The equality operators are left associative. Types used with equality operators are Int, Bool, and String.

The result is type Bool and the value is true if both expressions have the same value, and false otherwise.
expression == expression

Types used with equality operators are Int, Bool, and String. The result is type Bool and the value is false if both expressions have the same value, and false otherwise.
expression != expression

4.8 Boolean Operators

The boolean operators are left associative. Types used with the Boolean operators are Bool.

The result is type Bool and the value is true if both expressions are true and false otherwise.
expression && expression

The result is type Bool and the value is true if at least one of the expressions is true and false otherwise.
expression || expression

4.9 Assignment Operators

The assignment operators are right associative. The types used with the assignment operators are Int, Bool, String. Assignment stores the value of the second expression in the first expression, both expressions must be of the same type.
expression = expression

4.10 Object Creation

Objects are both declared and created within a single program in gridworld. Objects are declared through referencing its name as a new object:
new object testObject;

Object creation uses similar syntax, except replacing object with the name of the object being created
new testObject t;

4.11 Object Access

The attributes of an object can be accessed through dot notation. The type and value of the access is equivalent to the type and value of the accessed attribute:
object.attributeName

4.12 Array Creation

An array is initialized with the form [**expression, expression, ...**] where each expression must be of the same type. The result is an expression with its type being array, and the value being a reference to the array.

4.13 Element Access Operators

The elements in an array are accessed with bracket notation, with the form **arrayName**[**expression**]. The expression must be of type int and be within the bounds of the array. The result will be the element in the array at the index indicated by the expression.

5 Declarations

5.1 Type Specifier

The type specifiers are int, boolean, and string.

5.2 Variable Declaration

The variables can be initialized with a constant, literal value, or an expression as long as the type of the value and the type of the variable are the same. Variables are declared as:
typeSpecifier varName

5.3 Array Declaration

Arrays can be declared as: **typeSpecifier[] varName = [a, b, c];**

5.4 Function Declaration

Functions consist of a function header and a function body. The header takes the form of: **typeSpecifier funcName(typeSpecifier a, typeSpecifier b, ...)**

In the example above, the first type specifier indicates the return type, and the parameters are separated by commas and enclosed in parentheses. The function body is enclosed in brackets, for example, after the function header.

6 Statements

6.1 Expression Statement

Expression statements are the most common form of statement, which are simply of the form:
expression;

6.2 Conditional Statement

There are two basic forms of conditional statements:

1. **if (expression) statement**

2. **if (expression) statement else statement**

Any number of elif statements can be added after the if statement and before the else if there is one. A few examples are listed below:
if (expression) statement elif (expression) statement elif (expression) statement
if (expression) statement elif (expression) statement else statement

The expressions must be of type Bool and if the value is true, the statement directly after it will be executed, and once one is executed, any expressions afterwards will not be considered.

6.3 Loop Statement

The while statement has the form:
while (expression) statement

The statement is executed repeatedly as long as the value of the expression remains true. This expression is checked before each execution.

6.4 Break and Continue Statement

The break statement can be used to terminate a loop.
while (expression) statement
if (expression) break;
statement

The continue statement can be used in loops to terminate an iteration of the loop and begin the next iteration:
while (expression) statement
if (expression) continue;
statement

6.5 Return Statement

A function returns to its caller by the return statement. If an expression follows return, the value is given to the caller of the function and must be of the type specified by the function.
return expression;

7 Built-in Functions

7.1 print

The print function outputs text to either stdout or a file. The first parameter, being the output text, must be a string. The second parameter, also a string, specifies the filepath of output. The absence of the second parameter causes print to default to stdout.

```
// print example
print("Hello gridworld!");
print("Goodbye gridworld!", "output.txt");
```

7.2 read

The read function reads text from either stdin or a file. The parameter specifies the filepath of input. If no parameter is given, it reads from stdin.

```
// read example
input = read();
fileInput = read("input.txt");
```

7.3 object.give

The give function assigns an variable (referred to as an attribute) of the specified type to the object which called the function. The first parameter contains the type of the variable, e.g. (bool,int,string or list), and the second parameter the name of the variable.

```
// give example
dog.give(string, name);
new dog dog1;
dog1.name = "spot";
```

7.4 object.has

Checks whether or not an object has been given a specified attribute. Accepts a string and checks whether or not the object calling the function has a variable with that name.

```
// has example
if (dog1.has(name))
    print(dog1.name);
```

7.5 save/load

Due to the game-based nature of this language, a built in save and load function is essential for the user. Save creates a log file of actions that a user has made during runtime, while load reads a config file created by save and initializes the board in gridworld so as to recover a previous game state.

7.6 undo

Under the same logic as saving and loading, the step-wise operating nature of gridworld makes it possible to undo an action done by a user by reiterating the actions done before the current point minus one.

7.7 roll

Roll generates a random integer ranging from 1 to X, X being the integer parameter. This simulates the rolling of an X-sided die.

```
// roll example
print ("roll for initiative!");
print (roll(20));
```

8 Language Scope

8.1 Global Scope

All variables defined at the top-level of a program will be by default part of the global scope, and be visible, modifiable to the entire program. Be careful, to define the variables at the beginning of the program. If a variable is defined at the middle of the program, code written before this variable declaration can not access to this variable.

```
int sum;
int moverange;
boolean canMove (int a, int b){
    sum = a + b;//able to use varibale sum
    abs_distance = abs(a - b);//unable to use variable abs_distance
    if (abs_distance <= moverange) return true;
    else return false;
}
int abs_distance
```

8.2 Scope within a Function

Variables defined within the function are the local variables to that function, and it will be expired automatically when the function ends. Codes outside that function can not access or modify those variables.

```
int sum (int a, int b){
    return a+b;
}
a = a - b;//unable to use a, b outside the function sum()
```

9 Standard Library

9.1 Map

The map function takes two integers, to build an empty map with the size of n * m. The first integer indicates the length of the board and the second indicates the width. Once the map size is define, it can not be changed dynamically during runtime.

```
//map() sample
map myMap = new map(int, int);
```

9.2 Object

The object function takes six integers and a boolean, to build an object with the following attributes: position of x-axis on the map, position of y-axis on the map, health point, attack, defense, if the object or not, how far can the object move.

```
//object() sample
object objectRock = new object(posx=int, posy=int, hp=int, att=int, def=int,
    moveable=boolean, moverange=int);
objectRock.hp=2;
objectRock.positionx=3;
```

9.3 Attack

Attack function will calculate the moverange of the player, the attack point and the defense of the target, to return a boolean of whether player attacks or not and updates the attribuates of the objects. If attack action make sense, it return true, otherwise, it return false.

```
//attack() sample
attack(object, object);
```

9.4 Max, Min

The max and min function takes serveal integers, to return the maximum or the minimum one in the series of integers.

```
//max(), min() sample
max = (int, int, int);
min = (int, int);
```

10 Sample Game

10.1 Initialization

Before designing the game, the map and the objects on the map are needed to be intialized. Sample gives the initialization of the map size and some attribuates of objects.

```
//to initialize the map and objects
map rpgMap = new map(10, 10); //from 0~9
object objectRock1 = new object(9, 9, 5, 0, 0, false, 0);
object objectRock2 = new object(5, 5, 3, 0, 0, false, 0);
object objectPlayer = new object(0, 5, 10, 3, 2, true, 3);
```

10.2 Attack or Move

Some choices can be set during the game. In the sample game, it sets two move modes and describes the status changes of the objects after each movement.

```
//to implement attack or move behavior
while(true){
print("Choose attack or move: attack:1 ; move:2");
str behavior = read();
if (behavior == "1"){
    print("Choose the target: rock:1 ; rock:2");
    str target = read();
    if (target == "1"){
        if (!attack(objectPlayer, objectRock1)) continue;
    }
    else (target == "2"){
    }
}
}
if (behavior == "2"){
    print("input the point to go");
    int pointx = read();
    int pointy = read();
    if((pointx>=rpgMap.width) || (pointy>=rpgMap.height)){
        print("out of the map, please move again");
        print("input the point to go");
        pointx=read();
        pointy=read();
    }
    if(abs(pointx-objectPlayer.posx)+abs(pointy-objectPlayer.posy) >
        objectPlayer.moverange){
        print("out of the move range, please move again");
        print("input the point to go");
        pointx=read();
        pointy=read();
    }
    objectPlayer.posx=pointx;
    objectPlayer.posy=pointy;
}
}
```