**Green World Array**

The world is going to be represented by a 3-dimentional array. In the assembly code, that 3-dimentional array will actually be 1 single array of size width\*depth\*height. The functions that access the Green World need to give width, depth, and height coordinates, which will then need to go through a process of linearization described below, and the corresponding indices used within the world array will be generated automatically and methodically:

For now, the values for the width, depth and height will be hardcoded as constants in the generator code since there is no plan to allow the user to change the dimensions of the world. Let’s call them WIDTH, DEPTH, and HEIGHT respectively.

The first line in the Data Definition section of the assembly code will be the definition of the World Array.  
Let’s set the values of WIDTH, DEPTH, and HEIGHT to 20, 20, and 10 respectively.

The product of these 3 dimensions is 4000. That will be the size of our array. So, with these dimension values in mind, the first line in the data def section will always be:

VAR WORLD INT 4000 0

As for the content of the array, a convenient solution would have been to compute the address of objects (blocks and tubs being considered as objects with their attributes). Each integer value in that array would be either 0, or a number that signifies an address for an object with attributes. But since there is no way in that assembly language to compute the addresses and then refer to them within the assembly language, basically no pointers, the data for those objects will be stored directly in the array. Meaning that we need to represent each of these objects with all of their attributes using numbers and indices. Let’s first start with the coordinates of objects. Functions such as place() or uproot() will need to access the Green World Array, place() is given coordinates, and uproot will have to extract coordinates from the object given to it. These are not the only two functions accessing the Green World Array but are only given as examples.

To find the index of an object, the X, Y, and Z coordinates of that object will need to go through a process of linearization.

Index = (X-1)\*DEPTH\*HEIGHT + (Y-1)\*HEIGHT + (Z-1)

So, an object of coordinates (3,4,5) will have an index evaluated to: Index = (3-1)\*20\*20 + (4-1)\*20 + (5-1) = 864

After that, the cell within the array with the calculated index will be given a value based on the state of the object that is being assigned to it. For instance, a block might be represented by 100000000. But that cell alone is not enough, every cell filled by that object should take the same value.

As an example, let block called “a” have coordinates (X,Y,Z) evaluated to be (3,4,5). The block also has the dimensions (W,D) evaluated to be (2,2). In this case, W\*D array cells will be filled with the same value 100000000. Those cells will be all cells with coordinates ranging from X to X+W-1 and Y to Y+D-1. In this specific example. Cells 864, 884, 1264, and 1284 will take the value 100000000. The logic of the other functions will make sure these cells are not corrupted by other operations.

The function unmakeAyoub(), which unmakes a block, will use the blocks dimensions and coordinates to reach the same cells that it normally fills and set them all to their default value of 0.

**Blocks, Tubs, Earth, and Plants in the Green World Array**

Now let’s work on the formal representation of blocks, tubs, tubs filled with earth, and plants within tubs.

Blocks will be represented by a simple 100000000

An empty tub will be represented by a 200010000. The 5th digit is 1 by default since it will be read as the denominator for the volume of earth. The value of earth within a tub is always expressed as a fraction and ranges from 0 to 1. the next four digits after the in the tub representation are for its earth volume. Digits 2 and 3 are got the numerator, and digits 4 and 5 for the denominator. This places a limit on the granularity/precision of the representation of earth, since we’re only allowed 2 digits for the numerator and 2 digits for the denominator, the other digits will be ignored. Not that this does not apply to regular fractions used for arithmetic operations. In any case. A tub with 20/21 of its volume is filled with earth will be represented as 220210000. That leaves us with 4 digits, which will represent the height of a tree. 2 of which will remain unused, the last 2 digits will be the height of the tree, we will assume for now that the height can only range from 0 to 99.

Here is how we would represent a tub, half-filled with earth, and that has a plant with height 4 (which is twice its own height). Let’s assume that the coordinates for that tub are (4,7,1) and its dimensions (2,2,2)

The original index would be evaluated to: Index = 1200

Note that the size of the plant is only vertical and will apply to the whole surface of the tub it is placed on. An empty tub with dimensions (2,2,2) will have a volume of 8, but if it has a plant of size 4 planted within it, then the tub will have a volume of 16, as if its dimensions were (2,3,4)

The following cells will take the value 201020004 representing the tub: 1200, 1201, 1202, 1203, 1220, 1221, 1222, 1223, 1400, 1401, 1402, 1403, 1420, 1421, 1422, and 1423.

**Blocks, Tubs, Earth, and Plants in the Data Definition Section**

On top of being represented within the Green World Array. Objects will also have their variable representation in the data definition section. The logic of the primitive functions should make sure that the data for the object variables is always entirely consistent with the data found in the Green World Array.

**Variable declaration**

Variable declaration :- Dec Khikhi VARNAME; --> goes in DataDef section

VAR VARNAME INT 1

**Assignment**

**Assign literal value to variable** :- VARNAME :: 24; --> goes to Instructions Section

ASSIGN 24 VARNAME

If a variable is declared then initialized in a global scope then the initialization can be done in the DataDef section :-

Dec Khikhi VARMNAME;

VARNAME :: 6; --> goes to DataDef section

VAR VARNAME INT 1 6

**Assign expression to variable** :-

a :: 8+(7-b)\*2+z\*9; --> goes into Instructions secion

SUB 7 MLINJUY TEMP1

## 'MLINJURY' is the location for variable 'b'

MUL TEMP1 2 TEMP1

## 'PZAWFZS' is the location for variable 'z'

MUL PZAWFZS 9 TEMP2

ADD TEMP1 TEMP2 TEMP1

## 'PMLWKSI' is the location for variable 'a'

ADD TEMP1 8 PMLWKSI

## still need a way to automatically determine

## which temp variable to use to hold

## which value. Need to be more methodical.\*

**Constant Definition**

Constant definition :- $Hachem CSTNAME 90 --> goes to DataDef secion

CONST CONSTNAME INT 1 6

**Fraction Representation** Fractions are going to be split into two int variables, a numerator and a denominator :-

Dec Boubou fracname;

fracname :: :1.5; --> goes to DataDef secion

VAR POLZWZP1 INT 1 1

VAR POLZWZP2 INT 1 5

We hare going to assume that 'fracname' got hashed into 'POLZWZP' and that the '1' and '2' suffixes represent the numerator and the denominator respectively

**Control Flow Statements, Conditionals, and Boolean Expressions**

Ila(a < b) Idan{

## some code

} Sinon{

a :: a + b;

} --> goes to Instructions section

## for this example, 'a' and 'b' retain their

## same locations

JLESS PMLWKSI MLINJURY THEN1

## some code in the then statement

ELSE1 ADD PMLWKSI MLINJURY PMLWKSI

For boolean expressions, we were hoping that, like C, integers could be evaluated to booleans.

As in 0 for false, and non-zero values for true.

We were planning for boolean expressions to be evaluated to true or false and get an integer value from it (1 or 0)

but now that we read the GWAL documentation, there seems to be not support for integer values as substitutes for booleans.

Furthermore, it looks like GWAL does not evaluate boolean expressions for us, and also does not have native support for

AND, OR, and NOT operators. Meaning that complex boolean expressions involving comparisons and logical operators should be coded

as a succession for conditional jump statements following a methodical logic

There is no native support for the <= and >= operators, so those have to be turned into (< or ==) and (> or ==)

Some examples:

Ila (a < b or a == b) Idan {

## some code

} Sinon{

## some other code

}

## note that a < b or a == b is equavalent to a <= b --> goes to Instructions section

(a+b)\*(c+d\*c)