**PROJECT Part 1 – Language Design (draft)**

This document provides a brief description of the application for which you will design the simple programming language and some requirements that the language must satisfy.

**Application and Language**

The application is a three-dimensional world, as shown in Figure 1 below, called Green World (GW) whose goal is to allow a programmer to construct and change a world constructed of blocks, tubs and plants planted in the tubs. The language’s name is GWL (Green World Language).

**PARAMETERS and Objects**

The Green World characterized by the non-negative integer valued parameters *height*, *width* and *depth*. Each of these values can range from 0 to infinity but, realistically, they should be at least 1 or greater for anything interesting to happen. They are initialized at the start.

Within this world, there are four types of objects/stuff: **block**, **tub**, **earth**, **plant**.

|  |  |
| --- | --- |
| Chart, line chart  Description automatically generated  Figure 1: 3 dimensions of world | A **block** has size:   * Minimum: 1x1x1 * Maximum: *width* x *depth* x 1   A **tub** has size:   * Minimum: 1x1x1 * Maximum: *width* x *depth* x min(3, *height*)   **Earth** is measured by its volume, which is simplified to be less than or equal to the volume of the tub where it is placed. E.g., a 1x1x1 tub can contain a quantity of earth that ranges between 0 and 1. If the quantity of earth is not an integer, it is expressed as a fraction. E.g., 1/2 or 1/3.  A **plant** can have a size that ranges from 2 to infinity (theoretically). |

Also initialized at the start are the available quantities of each object: *qBlocks*, *qTubs*, *qEarth*, *qPlants*. See below for how these get used.

**Constraints that must hold between objects**

A plant’s size must be greater than the height of the tub it is placed in and cannot exceed   
*height*(**tub**) \* 3 if the tub is completely filled with earth or *height*(**tub**) \* 2 if the tub is not completely filled with earth.

A plant can only be put in a tub if the tub contains at least half its volume in earth.   
E.g., if a tub of volume 1 contains only 1/2 earth, then the plant cannot be higher than 2.

A block can be placed on top of a block. Any number of blocks can be put on top of a block.

A single block can be placed on top of a tub, but only if there is no plant in the tub, the block’s volume is no larger than that of the tub, and the tub is filled with earth.

A single tub A can be placed on top of tub B, but only if there is no plant in tub B and either tub A’s volume is no larger than that of tub B or tub B is completely filled with earth.

The total *width*, *depth*, and *height* of structures build using blocks, tubs and plants cannot exceed the parameters delimiting the world; otherwise, the program will crash.

**PRIMITIVE OPERATIONS**

The following primitive operations are available:

* *makeBlock*: It expects a *width* and a *depth*. It creates a block of the required size (remember *height* is 1) and returns it. Blocks magically appear from the initial supply if there are enough blocks left in *qBlocks*; otherwise, the operation fails.
* *unmakeBlock*: It expects an indication of the block to be unmade. It breaks up the block into blocks of size 1x1x1 and puts them back in *qBlocks.*
* *makeTub*: It expects a *width*, a *depth*, and a *height*. It creates a tub of the required size and returns it. Tubs magically get put together in the right shape if there are enough blocks left in *qTubs*; otherwise, the operation fails.
* *unmakeTub*: It expects an indication of the tub to be unmade. It breaks up the tub into tubs of size 1x1x1 and puts them back in *qTubs.*
* *place*: It expects either a tub or a block and an indication of where to put it in *w*, *d*, *h* space. It fails if the constraints on where you can put tubs and blocks are not respected.
* *makePlant*: It expects the size of the plant. It magically constructs it from *qPlants* if it has not been consumed and returns it; otherwise, it fails.
* *unmakePlant*: It expects an indication of the plant to be unmade. It removes it from the tub, breaks it up into unit size plants and returns them to the *qPlants* supply.
* *addEarth*: It expects an indication of the tub and the quantity of earth to be added to the tub. The earth magically appears, as long as the initial quantity *qEarth* has not been consumed; otherwise, the operation fails. The tub must have sufficient room left for that quantity of earth to be added to it; otherwise, the operation fails.
* *removeEarth*: It expects an indication the tub and the quantity to be removed. The earth magically disappears and gets added back to the initial heap *qEarth*. The operation fails if the quantity to be removed is not present.
* *plant*: It expects a plant and an indication of the tub where it should be planted. It fails if the constraints concerning plants, tubs and earth are not satisfied.
* *uproot*: It expects an indication of tub from which the plant is to be uprooted and removes it from the tub. It fails if there is no plant in the specified tub. Note that the plant must be unmade or the plant units it is made up of will be lost.

The primitive operations described above should be considered as built into GWL or as a library that is intrinsic to GWL, but you will need to write the code for them in the sense of generating the right VM language instructions for them.

More complex operations can be built by using them in combination with other language constructs.

**Other Language Elements**

From the above description of the application and language elements, you will need to determine what other elements you will need:

* Definitions/Declarations: For constants? For types (and, if so, what kind)? Variables?
* Statements: What kinds? Keep them as simple and as few as possible to reduce the amount of code generation you will have to do.
* Reserved Words: To use with definitions and declarations and statements
* Punctuation: What will you need?
* Operators: What kind and on what kinds of data types?
* Function call: Plan on having a way of defining functions whose parameters, if any, are passed by value, that return an optional result, and able to modify global variables.

**Program Examples**

As you design your language, you should try to develop some examples to see how programs look, while thinking about the concepts we are covering in class and through exercises.

An example of a simple program might be one that implements the following. Given a world with *width*=W, *depth*=D, *height*=H, where W=D are even integers, write a program that creates a structure of blocks and planters through the middle form *depth*  = 1 to *depth = D-1****.***

Chart

Description automatically generated

An example of a fairly complex program might be one that implements the following. Given a world with *width*=W, *depth*=D, *height*=H, where W=D=H, W is 11, write a program that creates a pyramidal structure. Figure 2 shows the structure that is build at the start of the program. Figure 3 shows what it becomes after the program modifies it.

|  |  |
| --- | --- |
| Chart  Description automatically generated  Figure 2. Structure built at the start of the program | Chart  Description automatically generated  Figure 3. Structure built at the end of the program |

**Submission & Grading**

The Team Captain submits to Jenzabar for the Team with a separate report on teamwork. Use the following filename: *TeamName*\_ProjectPart1 and *TeamName*\_ProjectPart1\_CapRep

The design should include a first draft of the Lexical Description (tokens), using regular expressions where it makes sense to, and Syntactic Description (grammar), using BNF of EBNF. Failure to use appropriate descriptive metalanguages results in points taken off.

The table on the next page serves as a checklist for you and also informs you of points attributed to each component.

You are also asked to submit two sample problems: a simple one and a more complex (but not too complex one). Please show what the task is graphically (as I did) and also provide the code written in the GWL you designed.

|  |  |
| --- | --- |
| **LANGUAGE ELEMENT CHECKLIST** | **POINTS** |
| **Lexical** | **10** |
| **Reserved words** | 3 |
| **Punctuation** | 2 |
| **Operators** | 3 |
| Arithmetic operators |  |
| Comparison operators |  |
| Logical operators |  |
| **Program Documentation (Comments)** | 2 |
| Single line |  |
| Multi line |  |
| **Syntactic** | **24** |
| **Definitions** | 8 |
| Constant Definition | 1 |
| Simple Types | 1 |
| Structured Type Definition | 3 |
| Variable Definition | 1 |
| Function Definition | 2 |
| **Expressions** | 6 |
| Arithmetic expressions | 2 |
| Comparison expressions | 2 |
| Logical expressions | 2 |
| **Statement Types** | 7 |
| Assignment (do not use = ) | 1 |
| Selection | 2 |
| Repetition | 2 |
| Function call (no return) | 1 |
| Function call with return | 1 |
| **Program Organization** | 3 |
| Main Program | 1 |
| Block structured | 2 |
| **Program Examples** | **6** |
| Simple | 2 |
| More Complex | 4 |
| **TOTAL** | **40** |