1 Introduction

Minecraft is a simple game where players can build, craft, explore and survive in an infinite sandbox world to their heart's content. With over 200 million copies sold, it is the best-selling video game of all time! Owing to that huge success, thousands of players design and build elaborate and beautiful structures in Minecraft, ranging from castles to cities to college campuses to a 1:1 scale model of the Starship Enterprise. Unfortunately, not everyone has the time or the talent to design or implement such grand builds. That's where MineBuilder comes in!

MineBuilder intends to redefine the way you build structures in *Minecraft: Java Edition.* Our easy-to-use language allows users to rapidly design and build structures in Minecraft without even having to move the character or press an in-game button! Structures are easily designed and built with various characters that correspond to actual blocks inside of the game. Instead of having to spend hours manually placing every block within the game, programmers will be able to easily design and manipulate structures out of ASCII characters in MineBuilder, and programmatically build them in seconds within the world of Minecraft!

2 Design Principles

MineBuilder is a statement based language that defines and builds structures in Minecraft. In designing the language, we wanted to balance ease of use with functionality. To that end, users can define 3D blueprints with ASCII characters in an intuitive, top-down approach, and then easily build them at certain coordinates!

A user's instructions will be parsed and interpreted in F#. Their instructions are transformed into Python scripts that send commands to a Minecraft server running on your machine, based on a Python API for the Raspberry Pi.

3 Example Programs

In order to run these examples, you have to own Minecraft, set up the Minecraft Python API, and be running a Spigot Minecraft server. Please visit this tutorial for help on setting up on Windows.

Example 0:

This example can be run with the following command from the minebuilder project directory:
dotnet run ../../examples/example-0.mc
Note: On Windows, make sure to change forward slashes to backlashes for proper pathing.

```
define woodStack as
{
     // this is the bottom layer made of wood
www
www
-
     // this is the top layer made of wood planks
WWW
WWW
-
}


// builds the previously defined structure at the given coordinates
build woodStack at (120,79,-244)

end
```

The expected output is the following:

```
Program successfully executed. Please check Minecraft to see results!
Additionally, check mine.py for the Python interpretation of your source code.
```

In Minecraft:

Example 1:

This example can be run with the following command from the minebuilder project directory:
dotnet run ../../examples/example-1.mc
Note: On Windows, make sure to change forward slashes to backlashes for proper pathing.

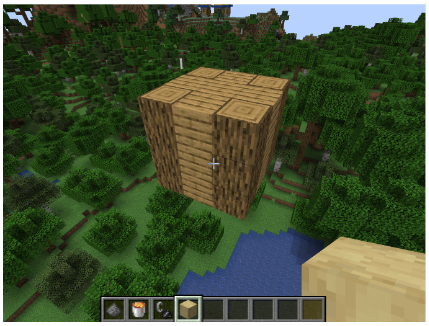
```
// defines a variable, storing the given blueprint in "struct."
// struct is a wooden box in this example
define struct as
{
    wWw
    WWW
    wWw
    -
    wWw
    WWW
    wWw
    -
    wWw
    WWW
    wWw
    -
}

// builds the defined structure at the given coordinates in Minecraft.
build struct (32,103,3012)
end
```

The expected output is the following:

```
Program successfully executed. Please check Minecraft to see results!
Additionally, check mine.py for the Python interpretation of your source code.
```

In Minecraft:



Example 2:

This example can be run with the following command from the minebuilder project directory:
dotnet run ../../examples/example-2.mc
Note: On Windows, make sure to change forward slashes to backlashes for proper pathing.

```
define pyramid as
{
    //load the blueprint for a pyramid
    load pyramid.mpb
}

// builds the defined structure at the given coordinates in Minecraft.
build pyramid (-23,-30,-12)
end
```

The expected output is the following:

```
Program successfully executed. Please check Minecraft to see results!
Additionally, check mine.py for the Python interpretation of your source code.
```

In Minecraft:

Example 3:

This example can be run with the following command from the minebuilder project directory:
dotnet run ../../examples/example-3.mc
Note: On Windows, make sure to change forward slashes to backlashes for proper pathing.

```
// defines a variable, storing the given blueprint in "boulder"
define boulder as
{
    .C.
    CCC
    .C.
    _
    CCC
    CCC
    CCC
    _
    .C.
    CCC
    .C.
    _
}

// defines a variable, storing the given blueprint in "house"
define house as
{
    //load the blueprint for a house
    load house.mpb
}

// repeatedly builds the given structures at the given Minecraft coordinates
// specifies the distance apart, and the direction of subsequent repetitions

repeat boulder starting at (100,80,-400) for 20 times, 3 apart in y direction
repeat house starting at (0,0,0) for 20 times, 10 apart in x direction
end
```

The expected output is the following:

```
Program successfully executed. Please check Minecraft to see results!
Additionally, check mine.py for the Python interpretation of your source code.
```

In Minecraft:

4 Language Concepts

In MineBuilder, a user defines blueprints of Minecraft structures that are later built using **build** or **repeat** commands. Blueprints are 3D structures, represented as a sequence of 2D slices made up of lines of specific ASCII characters corresponding to blocks in Minecraft. These horizontal slices are delimited by "-". Using these blueprints, a user can build or repeat any structure to their heart's desire in the world of Minecraft.

In order to communicate with the game, programs interface with Python and Spigot Minecraft servers running on the machine. Therefore, in order to run these types of programs, you will need to install the Minecraft Python API, and make sure you are running a properly configured Spigot Minecraft server. For more information on how to set up the Minecraft Python API, visit this tutorial.

5 Syntax

```
WHERE + = one or more
WHERE * = zero or more

<program>        ::=  <command>+
<command>        ::=  <define><nl>+
                      | <build><nl>+
                      | <repeat><nl>+
                      | <end>
<define>         ::= define <name> as<nl><blueprint>
<build>          ::= build <name> at <coords>
<repeat>         ::= repeat <name> starting at <coords> for <num> times,
                       <num> apart in <dir> direction
<end>            ::= end
<name>           ::= <character>+
<character>      ::=  any in {a..z,A ...Z}
                     | <num>
                     | <hyphen>
                     | _
<hyphen>         ::= -
<blueprint>      ::=   {
                          <slice>+
                       }
                     | {
                         load <filename>
                       }
<dir>            ::= any in {x,y,z,X,Y,Z}
<filename>       ::= <character>+ .mpb
<slice>          ::= <line>+ <nl> <hyphen>
<line>           ::= <block_chars>+ <nl>
<block_chars>    ::= any in {block character set}
<coords>         ::= (<num>,<num>,<num>)
<num>            ::= any in Integers
<nl>             ::= \n
```

6 Semantics

1. The **define** statement, which defines a name as a certain blueprint. Its usage is as either of the following:

```
define name as
{
    load example.mpb
}
```

```
define name2 as
{
    www
    WWW
    -
}
```

Both invocations adds the name-blueprint tuple to the global environment. The left loads example.mpb, a file, and associates it with name, while the right simply associates the given blueprint with name2. In the Abstract Syntax Tree, they are stored as a **Define** of (string * Blueprint)

2. The **build** statement, which builds the blueprint associated with a name at the given coordinates in Minecraft. Its usage is the following:

```
build name at (0,100,200)
build name2 at (-2,-2,-2)
```

The **Build** command connects with the Minecraft client and builds the blueprint at the given coordinates. The first builds name at (0,100,200), and the second builds name2 at (-2,-2,-2). In the AST, they are stored as a **Build** of (string * Coords)

3. The **repeat** statement, which repeats the blueprint associated with a name at the given coordinates in Minecraft for a given number of times, with a given distance, in the given direction. Its usage is the following:

```
repeat name starting at (0,100,200) for 20 times, 3 apart in y direction
repeat name2 starting at (-2,-2,-2) for 5 times, 10 apart in x direction
```

The **Repeat** command connects with the Minecraft client and repeats the blueprint, starting at the given coordinates, and continuing for the given number of times in the given direction, with the given distance separating each instance. The distance is measured from the widest points of the blueprint in the given direction. The first repeats name for 20 times, 3 apart in the Y direction starting at (0,100,200). The second repeats name2 for 5 times, 10 apart in the X direction starting at (-2,-2,-2) In the AST, they are stored as a **Repeat** of (string * Coords * int * int * char)

4. The **end** statement, which signals that the program is over and tells the interpreter to build and run the Python file. Its usage is the following:

```
end
```

The **End** command is what signals to the interpreter that the program is done, and actually writes the output of the previous build and repeat commands to Python. Then, it will run the Python script to build the delineated structures in the game, and close execution. In the AST, they are stored as an **End**.

Others. In order to use the above commands, the following two data types are used:
1. The three data types that make up the **Blueprint** data type are of the structure:

```
Blueprint of Slice list
Slice of Line list
Line of char list
```

All together, these three represent the three dimensions of the Blueprint. Each Blueprint represents the 3D encoding of the given structure. Each Slice represents a 2D X-Z slice of the given structure. Each Line represents a 1D x-dimension block line of the given structure, encoded as characters.

2. The **Coords** data structure is simply a 3-tuple of three integers.

7 Remaining Work

Much of the desired functionality of our language has already been implemented, as users can define blueprints, build them, and build them repeatedly.

To that end, the following are some *stretch goals* we would have liked to implement given infinite time and resources:
1. A way to combine blueprints in an intuitive way, such as defining them in terms of each other (e.g. a house defined as a combination of a special roof, walls, floor, and ceiling)

2. A way to alter the dimensions of the blueprints (such that they could be built using any direction of 2D slices including X-Y and Y-Z)

3. A refactoring that would eliminate the need to use the Python API - this is the most difficult of the three, as it would involve creating a user mod for Minecraft to directly interface with our program.

4. If the following refactoring was completed, an interactive REPL that built structures in realtime, allowing instant feedback and troubleshooting.