# Cellmata language guide

Group: d409f19

May 17, 2019

# Contents

# 1. Language guide

A Cellmata program has two parts. In the top is a world-declaration which specifies the grid and rendering properties. Following the world declaration are the components of the cellular automata, like states and neighbourhoods, along with any constants and functions.

## 1.1 World declaration

The world declaration specifies the various properties of the simulation.

Code block 1.1: World declaration example

```
world {
    size = 10 [wrap], 20 [edge];
    edge = DEAD;
    tickrate = 10;
    cellsize = 4;
}
```

The example above specifies a 2-dimensional world, that is 10 cells wide in the X axis and 20 cells high in the Y axis. As seen for the X axis, `[wrap]` can be specified to make the edge wrap around to the other side of the board. On the other hand, cells below Y=0 and above Y=19 are always in the Dead state due to `[edge]` after the Y dimension and the specified `edge = Dead;`. The edge option is only needed, if there's an `[edge]` dimension.

The option called `tickrate` specifies the number of updates per seconds in hertz and the option called `cellsize` is the size of each cell when rendered.

Note that within the world-declaration the `size`-declaration must be declared first, and optionally, if any dimension has an edge defined, the edge option must immediately follow the size option. The `tickrate`- and `cellsize`-options are optional and can be placed in any order, as long as they follow the size- and edge-declaration.

## 1.2 Main body

After the world-declaration follows a series of state-, constant-, neighbourhood-, and function-declarations.

### 1.2.1 State declaration

A state block declares the existence of a state and the logic that should be run to update a cell in the given state each tick.

Code block 1.2: State declaration example

```
state Dead (255, 255, 255) {
    if (randi(0, 10) == 0) {
        become Alive;
    }
}
```

The above example declares the existence of the state called `Dead` and a small amount of logic that changes the cell's state to the `Alive` state at a 1/10 chance each tick.

Each state has a colour assigned to it, to show during the rendering of the CA. This colour is in RGB-notation and is declared in parentheses after the name of the state. Below is a lookup table that has the most common colours in RGB-notation.

| | |
|---|---|
| Black | (0, 0, 0) |
| Blue | (0, 0, 255) |
| Brown | (165, 42, 42) |
| Cyan | (0, 255, 255) |
| Gold | (255, 215, 0) |
| Green | (0, 128, 0) |
| Gray | (128, 128, 128) |
| Maroon | (128, 0, 0) |
| Orange | (255, 165, 0) |
| Pink | (255, 192, 203) |
| Purple | (128, 0, 128) |
| Red | (255, 0, 0) |
| White | (255, 255, 255) |
| Yellow | (255, 255, 0) |

### 1.2.2 Neighbourhood declaration

Neighbourhoods are declared as a list of relative coordinates in the main body. Code-block 1.3 shows a formal neighbourhood declaration.

Code block 1.3: Neighbourhood declaration example

```
neighbourhood diagonalNeighbours {
    (1, 1), (1, -1), (-1, 1), (-1, -1)
}
```

When a cell is updated, the neighbourhood name will be mapped to an immutable array of states. The states in the array are the states of the cells located relatively to the updating cell. The lookup-expression can be used to get the state of a specific cell in the neighbourhood, see 1.4. The order of the states in the array depends on the order declared in the formal neighbourhood declaration.

Code block 1.4: Neighbourhood lookup example

```
state copyAbove (50, 50, 50) {
    become diagonalNeighbours[0];
}
```

Other common ways to use neighbourhoods are the built-in `count` function and pattern matching by comparing the neighbourhood to an array. See both in use in code-block 1.5. Section 1.3 will elaborate on built-in functions.

Code block 1.5: Other common uses of neighbourhoods

```
state monster (255, 120, 170) {
    if (diagonalNeighbours == {human, human, monster, monster}) {
        become warzone;
    } elif (count(monster, diagonalNeighbours) < 2) {
        become human;
    }
}
```

### 1.2.3 Constant declaration

Constant declarations specify global constant values that can be used to tweak various parameters of cell transition logic more easily. Note that constant-declarations are only allowed in the main body.

Code block 1.6: Constant declaration example

```
const depth = 10;
```

### 1.2.4 Functions

Helper functions can be declared using a function-declaration.

The function-declaration in code-block 1.7 shows a trivial function named `getFourtyTwo`. Functions are prefaced by the keyword `function`. This `getFourtyTwo` takes no arguments and returns and integer, in this case, 42.

Code block 1.7: Trivial function declaration example

```
function getFourtyTwo() int {
    return 42;
}
```

The function-declaration in code-block 1.8 shows a simple function, named `increment`, which has one argument called `value`, and it returns that value plus one. Code-block 1.9 shows how to call this function and the return-value in the comment following the call.

Code block 1.8: Function declaration example

```
function increment(float value) float {
    return value + 1;
}
```

Code block 1.9: Function call example

```
x = increment(41); // Variable x is 42.0 after executing function-call
y = increment(41.5); // Variable x is 42.5 after executing function-call
```

In Cellmata function calls are expressions and must always return something.

### 1.2.5 Types

In Cellmata there are the following types:

- `int` : Integers, 1, 2, 4, -10, 1231
- `float` : Decimal numbers, 1.0, 12.3, 3.1415
- `bool` : Booleans, either 'true' or 'false'
- `state` : cell states
- `neighbourhood` : local neighbourhoods

Cellmata will implicitly convert integers to floats where needed. The built-in functions `ceil()` and `floor()` which rounds up and down to the nearest int can be used to convert from float to int.

Arrays are sequences of values of the same type. They are constructed by surrounding expressions with curly-brackets. The lookup-expression can be used to fetch a specific value as seen in 1.10.

Code block 1.10: Arrays example

```
let arrOfInt = {5, 10, 2 * 12};
let firstInt = arrOfInt[0]; // set firstInt to 5
```

The type of an array is written as the basetype followed by a pair or square-brackets. This is relevant for functions, where you write the type of arguments and the return type. See 1.11 below.

Code block 1.11: Arrays type example

```
function secondInt(int[] list) int {
    return list[1]; // return the second element
}
```

## 1.3 Built-in functions

Cellmata has multiple built-in functions which are expected to be useful in most of the computations needed for a given state's logic. The function signatures are listed below:

- `count(state s, neighbourhood n) int`
- `randi(int min, int max) int`
- `randf(float min, float max) float`

- `absi(int value) int`
- `absf(float value) float`
- `floor(float value) int`
- `ceil(float value) int`
- `root(float value, float root) float`
- `pow(float value, float exponent) float`

Some functions has both an integer and float variant. E.g. `randi` returns a random integer, and `randf` returns a random float.

Flooring and ceiling-function are available, which can be used for converting a float to an int, e.g., consider the function `roundFloat` in code-block 1.12.

Code block 1.12: Rounding function example

```
1 function roundFloat(float value) int {
2     // Rounds 'value' to nearest integer with tie-breaking towards positives
3     return floor(value + 0.5);
4 }
```

Power- and root-function are also available for more complicated computation.

The final built-in function; `count`, is likely the most useful and significant feature of Cellmata, as it allows the programmer to, given a defined neighbourhood and a defined state, easily get the count of neighbouring cells with the given state. Consider the state-declaration in code-block 1.13. At line 2, a local variable is assigned the output of the function-call `count(human, area)`, which counts the number of `alive`-states given the defined neighbourhood called `alive`.

Code block 1.13: Count function example

```
1 state human (0, 0, 0) {
2     let numberOfHumansInArea = count(human, area)
3 }
```

## 1.4 Code syntax

### 1.4.1 If statements

If-statements are used to create branching based on condition(s).

Code block 1.14: Syntax for an if-statement

```
if (<expr>) {
    <stmts>
} elif (<expr>) {
    <stmts>
} else {
    <stmts>
}
```

Note that unlike languages like C, it is disallowed to omit curly brackets around the body of a if-statement. See the following example of such an omission of brackets:

```
if (x > 0) return 42; // Not allowed in Cellmata!
```

### 1.4.2 Assignment statement

In Cellmata there is type inference, which means that the type of the variable can be omitted when the variable is declared. The variable automatically gets the correct type, inferred from the value given in the context. The keyword `let` is used to declare new variables. In code 1.15 line 1 a variable is declared, and in line 2 given a new value.

Code block 1.15: Syntax for an assignment statement

```
let <ident> = <expr>;
<ident> = <expr>;
```

In code 1.16 line 1 a variable is declared to be 4.20, and as such the variables gets the type float, but this is never explicitly declared, this variable is later assigned a new value. Likewise, on line 3, `y` is declared to be true, and therefore `y` is a boolean.

Code block 1.16: Syntax for an assignment statement

```
let x = 4.20;
x = 8.5;
let y = true;
```

### 1.4.3 Loops

The for-loop is the only iterative construct in Cellmata. It consist of two assignments; a initial assignment, and a post-iteration assignment, an expression, which is the condition, and finally the block of statements, which should be iterated. See code-block 1.17 and 1.18 for examples. The first assignment is executed before the loop and any variable declared here persist during the for-loop construct. The loop iterations as long as the condition is true. The second assignment is executed after each iteration.

Code block 1.17: Syntax for a for-loop statement

```
for (<assignment>; <expr>; <assignment>) {
    <stmts>
}
```

Code block 1.18: Syntax for a for-loop statement in use.

```
for (let i = 0; i < 8; i = i + 1) {
    if (Moore[i] == Dead) {
        become Dead;
    }
}
```

### 1.4.4 Return statement

The `return` statement is used only in functions that you declare yourself. The expression after the `return` statement is sent back to the call to the function. The type of the expression must match the return type declared in the function declaration, see section 1.2.4. If a `return`-statement is met in a function, the code following the statement will not be run.

Code block 1.19: Syntax for a return statement

```
return <expr>;
```

In code 1.20 the int 5 is returned.

Code block 1.20: Syntax for a return statement

```
return 5;
```

### 1.4.5 Become statement

The `become` statement is used only inside `state`-blocks, to allow a cell to change state. When the `become` <state> is met, the cell will change to the given state. Code after the `become`-statement will not be run.

Code block 1.21: Syntax for a become statement

```
become <state>;
```

### 1.4.6 Expressions

The following code-block 1.22

Code block 1.22: Syntax for expressions

```
(<expr>) // Parenthesies, for ensuring computation precedence
<expr> [<expr>] // Array lookup, the first expression must evaluate to an array, and
 ↪  the second must evaluate to an integer
! <expr> // Negation, the boolean inverse of the given expression
-<expr> // Negation, the numeric inverse of the given expression
/* Binary infix operators follows - this means that the two elements are on either
 ↪  side of the operator symbol, e.g. 1 + 2 */
<expr> % <expr> // Modulo operator
<expr> / <expr> // Division operator
<expr> * <expr> // Multiplication operator
<expr> - <expr> // Subtraction operators
<expr> + <expr> // Addition operators
<expr> < <expr> // Less than operator. Returns true if first expression is strictly
 ↪  less than the second expression
<expr> <= <expr> Less than or equal operator. Returns true if first expression is
 ↪  less or equal to the second expression
<expr> > <expr> // Greater than operator. Returns true if first expression is
 ↪  strictly greater than the second expression
<expr> >= <expr> // Greater than or equal operator. Returns true if first expression
 ↪  is greater or equal to the second expression
<expr> == <expr> // Equality operator, returns true if the expressions are equal -
 ↪  works for both boolean and numeric values
<expr> != <expr> // Inequality operator, returns true if the expressions are not
 ↪  equal - works for both boolean and numeric values
<expr> && <expr> // Boolean and operator, returns true, if both expressions
 ↪  evaluates to true
<expr> || <expr> // Boolean or operator, returns true if either expression is true
```

```
{<expr>, <expr>} // Array initialisation, must be of same type or implicitly
 ↪  converted
```

### 1.4.7 Example Program: Wireworld

In code-block 1.23 a famous cellular automata called Wireworld is shown, written in Cellmata. Wireworld is used to simulate electricity running through wires. I.e. Wireworld requires a custom start configuration to be interesting, but it still shows how a complete cellular automata looks like in Cellmata.

Code block 1.23: Wireworld in Cellmata

```
world {
        size = 50 [edge], 50 [edge];
        edge = Empty;
        tickrate = 6;
        cellsize = 6;
}

neighbourhood mooreNeighbours {
    (-1, 1), (0, 1), (1, 1),
    (-1, 0),         (1, 0),
    (-1, -1), (0, -1), (1, -1)
}

state Empty (0, 0, 0) {

}

state ElectronHead (100, 120, 255) {
        become ElectronTail;
}

state ElectronTail (255, 140, 50) {
        become Conductor;
}

state Conductor (220, 210, 50) {
        let electrons = count(ElectronHead, mooreNeighbours);
        if (electrons == 1 || electrons == 2) {
                become ElectronHead;
        }
}
```