Documentation

AMETHYST

Compiled language framework for Minecraft Datapacks and Resourcepacks

Fabian Hummel

Preface

Datapacks and Resourcepacks are Minecraft's way of scripting custom logic and content into the game using configuration and script files. Since the very beginning, the Minecraft Function language (MCFunction or MCF in short) has been a very verbose and bloated language with often different ways to achieve the same thing which comes at the cost of a relatively high learning curve.

While the MCF language itself can be considered fairly powerful and extensive, it lacks modern syntax which you would expect from a language of today's standard. Resourcepacks on the other hand are relatively easy to write and understand, but very tedious to maintain because of their nature by requiring dozens of files in a confusing amount of different directories.

Amethyst simplifies these things by abstracting the configuration and logic behind a modern, high-level language specifically designed to seamlessly integrate with the game's mechanics. It includes a built-in preprocessing engine to maintain thousands of lines of code at once, and even optimises your code on the fly as best as it can to gain near-native command execution.

Table of contents

1. Introduction

1.1 Installation

Amethyst comes in prebuilt binaries for all major Operating Systems and can be downloaded from the Github repository. Once installed, you can start creating Amethyst projects right away.

1.2 Usage

1.2.1 Configuration

Projects can be created by creating an `amethyst.toml` (main configuration file) and a `src/` folder where all the project code will be in. Files end in `.amy` endings. All other files will be ignored for compilation (with some exceptions like resources)

```
minecraft = "/path/to/.minecraft" # if .minecraft is not at the
    default location

[datapack]
name = "playground" # project name (resulting folder name)
namespace = "playground" # datapack namespace (defaults to the
   project name)
```

```
description = "Test project for the amethyst compiler" #
   datapack description
pack_format = 48 # specific pack format (defaults to the latest
   )
output = "New World" # folder name of the world where the
   datapack will be compiled into

[resourcepack]
name = "playground"
namespace = "playground"
description = "Resources for the test project"
pack_format = 34
```

1.2.2 Compilation

After the project is configured, the compiler can be either ran once to compile the entire project, or in **watch mode** to continuously compile the project whenever a source file changes by invoking the compiler with the flag `-w` or `--watch`.

When the compiler has finished creating the datapack or resourcepack, you can `/reload` the datapack in Minecraft or press `F3 + T` to reload the resourcepack.

1.2.3 Building for production

The data- and resourcepack can be compiled to a production build (`.zip` file) that automatically drops all `debug` statements / comments and minifies the output code to the bare minimum to gain performance and to reduce the filesize as much as possible. To create a production build, invoke the compiler with the flag `-r` or `--release`.

1.3 Quick feature tour

1.3.1 Datatypes and Type safety

In Amethyst, variables are statically typed, with some exceptions to dynamic data structures like dynamic arrays. Thankfully, types are inferred most of the time, so you don't have to worry remembering the types everytime.

```
var x = 5; # x will be of type integer
x = "Hello"; # error: wrong assignment
```

```
var y: string; # ok, type is defined at declaration
y = "Amethyst Shard";
```

```
var x; # error: type cannot be inferred
```

1. **Strings** `string "MyString"`
2. **Integers** `int 1537879` - Integers do not have decimal precisions and range from **-2.147.483.648** to **2.147.483.647**.
3. **Decimals** `dec 8294.245` - Decimals have a 3-digit precision due to scoreboards being unable to store decimal numbers and therefore only range from **-2.147.483**,648 to **2.147.483**,647.
4. **Booleans** `bool true`, `false`
5. **Typed Arrays** `<type>[]` - Arrays with a fixed type can only contain elements of that exact type.
6. **Dynamic Arrays** `array` - Arrays that can contain any elements of any datatype. Keep in mind that working with values inside such arrays have limited functionality. See *Dynamic Arrays* for detailed explanation.

7. **Typed Objects** `<type>{}` - Objects that hold key-value pairs of elements with a fixed datatype. Keys are always strings.

8. **Dynamic Objects** `object` - Objects that hold key-value pairs of any data type. Keys are always strings. Note that working with values inside such objects have limited functionality. See *Dynamic Objects* for detailed explanation.

1.3.2 Functions

All functions defined in Amethyst are essentially normal function as you'd expect them from normal MCF code, with some little extras.

```
function my_function(message: string) {
    debug message;
}

[on_initialize]
function init() {
    my_function("Hello Minecraft World!")
}
```

We can conclude that functions have parameters that can be used to pass values to different parts of the logic.

Also note the `[on_initialize]` attribute. This marks the function as such to run once at the start of the game. It is simply added to the `load.json` and runs after Amethysts own initialization code. Another common attribute is `[on_tick]`, which marks a function to run every frame and can even be mixed with `[on_initialize]`.

To use functions, same with variables, they need to be in scope[1] and called with parenthesis

[1] An area where a specific variable or function can be accessed. The usage of such needs to be either above on the same level as its definition or one of its ancestors.

like in most programming languages. Arguments can be passed via a comma-separated list between the parenthesis.

The return value can immediatetly be used to assign a variable or print the result like an ordinary value.

1.3.3 Namespaces

To reduce clutter within the codebase itself, Amethyst uses namespaces to group functions and logic into smaller, hierarchical groups. Create a namespace with `namespace <name>`.

file_1.amy

```
1  namespace outer {
2      function a(): string {
3          return "Hello from outer!";
4      }
5
6      namespace inner {
7          function b() {
8              debug a(); # Hello from outer!
9          }
10     }
11 }
```

Namespaces create a new scope, just like functions, which means that function `b()` can call function `a()` without needing to prefix it with anything, because the function is defined in an ancestor scope.

If function `a()` were to call `b()`, it would need to invoke it by using `inner::b()`.

Namespaces can also be defined as many times as needed, and even across multiple files. To call function `b()` from another file, while still being in namespace `outer`, we would again use `inner::b()`, because this is where the function is defined relative to namespace `outer`.

file_2.amy

```
namespace outer {
    debug inner::b(); # Hello from outer!

    # namespace b and function inner() defined in another file!
}
```

Sibling namespaces however have also access to each other, which means that a top-level namespace `outer_sibling` has full access to `outer` using `outer::<whatever>`

```
namespace outer_sibling {
    debug outer::a(); # Hello from outer!
}
```

1.3.4 Selectors

Amethyst takes Minecraft's Selectors to a new level by allowing the reuse of Selectors throughout the program, do calculations upon them and mix them with variables to allow for comprehensive logic.

The idea of the @-Sytax with brackets is kept the same, but nicely integrated into the Amethyst language.

```
record x = 5;

```

```
var players = @a[
    record:x=..10,
    distance=100..
];

foreach player in players {
    tellraw player "My score: " + x[player];
    # or tellraw @s "My score: " + x[@s];
}
```

Also note the `foreach` statement that is used to iterate over the players. In this case, it purely serves as syntactic sugar and is converted internally to the traditional `execute as @a[...] run [...]` command. Usually, `foreach` is rather used to iterate over a data source like arrays or objects. More on that in section 1.3.5

1.3.5 Loops

Loops can be used to iterate over a number of different data structures like arrays, objects, strings and players. Iteration may happen through many different ways, namely `for`, `while` and `foreach`.

Iterating over an array of elements with a classic for loop

```
var arr = ["Hello", "Minecraft", "World"];
for (var i = 0; i < arr.length; i++) {
    debug arr[i]; # Hello
}                 # Minecraft
                  # World
```

Iterating over elements of an object with a foreach loop

```
var obj = {"weather":"raining", "humidity":60, "cloudy":true};
```

```
foreach (key, value) in obj {
    debug key + ": " + value;
}
```

1.3.6 Variables and Player-specific data

In Minecraft there are two types of variables. On one hand, variables can be allocated to *central data sources*[2], and are created using the `var` keyword. On the other hand, variables can also be bound to players, so each player holds their own value for a specific variable. In Amethyst, these types of variables are called records and can be created using the `record` keyword.

When creating a record, keep in mind that the default value will be assigned to each player in the world once, or whenever they *join or leave the game*[3]. When assigning a record during execution of the program, the value will be set for all players immediately.

Increase points for a player that kills another player

```
ADVANCEMENT player_killed_player {
  "criteria": {
    "requirement": {
      "trigger": "minecraft:player_killed_entity",
      "conditions": {
        "entity": {
          "type": "minecraft:player"
        }
      }
    }
  }
```

[2]These can be global storages or scoreboards. In the case of Amethyst, these would be the `amethyst :internal` storage and the `amethyst` scoreboard, where all normal variables are stored.

[3]Special events on when to reassign the default value can be configured for a record using the attributes `[assign_on_join]` and `[assign_on_leave]`. This is useful to reset e.g. points in a minigame whenever a player leaves the game.

```
}

record points = 0;

[player_killed_player]
function handle_player_killed_enemy() {
    points[@s]++;
}
```

Note the use of the attribute `player_killed_player`, which is an advancement defined earlier that grants a player that has killed another player. This advancement can then be assigned to a function via an attribute, which adds that function to the rewards list and will execute whenever that event triggers.

Also note that the points only increase for the player that has actually killed an enemy using the `@s` selector. Records can be combined with a selector to change a value only for the entities matching the selector.

1.3.7 Basic Commands

Thankfully, adapting to Amethyst is fairly easy due to common commands having familiar syntax with MCF.

Execute The execute command has all the functionality that's expected from this command, with some neat extra features to provide better integration with Amethyst. Things like **block bodies** or **functions** can be used after the `run` keyword instead of traditionally only one following command. Therefore, syntax like this is possible:

```
execute as @a at @s run {
    setblock ~ ~-1 ~ "diamond_block";
    tellraw @s + " placed a diamond block!";
```

```
};
```

```
function give_coin_and_say_thanks(amount: int) {
    give @s "gold_nugget" amount;
    tellraw @p[tag="Admin"] @s + " thanked you for " + amount +
        " coin(s)";
}

execute as @a run give_coin_and_say_thanks(rand_int(10));
```

1.3.8 Data Resources

As already used earlier in an *example snippet*, Amethyst allows for inline declaration of data resources such as **Advancements, Predicates, Loot Tables** and much more you would find in any ordinary datapack. The same applies to resourcepacks, where we can declare **textures, models**, and more right in the Amethyst project.

Resources are declared with uppercase letters "`ADVANCEMENT`" or "`PREDICATE`" which means they are *compile-time constants* and let you preprocess them however you like to save important time[4].

With the use of `FOR` and `YIELD` you can save hours of tedious work of copying files or code over and over again with the risk of making simple mistakes by letting the compiler do the work for you.

```
[override]
RESOURCE models/item/carrot_on_a_stick {
  "parent": "item/generated",
  "textures": {
```

[4] In this case, compile-time constants allow the creation a template resource with variables and automatically generate multiple variants of that resource during compilation.

```
    "layer0": "item/carrot_on_a_stick"
  },
  "overrides": YIELD (VAR frame = 0; frame < 20; frame++) {
    "predicate": { "custom_model_data": frame },
    "model": "item/magic_wand/equip_" + frame
  }
}
```

```
FOR (VAR variant = 1; variant <= 10; variant++) {
    [override]
    RESOURCE "models/block/dirt_" + variant {
      "parent": "block/cube_all",
      "textures": {
        "all": "block/dirt_" + variant
      }
    }
}
```

Traditional resource files (`.json` files) can still be placed in the project under the directory where they would normally go in a data- or resourcepack and will be copied to the output directory as usual. However, keep in mind that these files do not support preprocessing natively through Amethyst.

This way, you can even add your own `.mcfunction` files to the datapack if you require to keep some functions native or want to include a datapack library.

```
amethyst.toml
/src
|   /data
|   |   /my_pack
|   |   |   /loot_tables
|   |   |   |   /my_loot_table.json
```

```
|   |   /included_library
|   |   |   /functions
|   |   |   |   /lib_code.mcfunction
|   main.amy
```

1.3.9 Preprocessing

Speaking of preprocessing, it's also possible to define compile-time constant functions with the `FUNCTION` keyword. It works like a regular function, but is evaluated during compile-time and is mostly used to generate boilerplate code.

Additionally, the preprocessing engine allows for more complex control flow with `IF`, `ELSE` and loops (`FOR`, `WHILE` and `YIELD`).

However, note that variables used within preprocessing need to be declared as compile-time as well with an uppercase `VAR`. This is in order to distinguish runtime variables from compile-time ones.

1.3.10 Debugging

Currently, Amethyst does not yet feature a way of dynamically debugging a program (Although I do have plans for that using a vscode-extension), but can still be done using the `debug` keyword that logs a message to all players on the server (therefore should not be used in production).

Furthermore, the debug keyword also syntax-highlights datatypes such as objects, arrays and numbers and make debugging a little easier on the eyes (at the cost of performance).

```
var x = 10;
debug x; # [DEBUG]: 10
```

2. Examples

2.1 Fibonacci Sequence

The fibonacci sequence is a classic example to demonstrate different aspects and capabilities of a programming language. Therefore, this algorithm can be used to show off basic syntax of a language and get a first impression. In the case of Amethyst, it does not use any special features that have anything to do with Minecraft, but is still a good example for recursion and mathematical expressions.

```
1 function fibonacci(n: int): int {
2     if (n < 2) { return n; }
3     return fibonacci(n - 1) + fibonacci(n - 2);
4 }
5
6 for (var i = 0; i < 10; i++) {
7     debug fibonacci(i); # 0 1 1 2 3 5 8 13 ...
8 }
```

3. Reference

3.1 Datatypes

Some type of values are stored in scoreboards, while others are better suited for storages.

1. **Strings** `"MyString"` are stored in storages as normal strings.
2. **Integers** `1537879` are stored natively in scoreboards.
3. **Decimals** `8294.245` are stored with 100x scale in scoreboards (to preserve decimal precision).
4. **Booleans** `true, false` are stored as 0 or 1 inside scoreboards. Altough storages support boolean values natively, it's easier and faster to do calculations with 0 and 1 like toggling between true and false with a single command.
5. **Dynamic Arrays** `[]` are stored in storages with special formatting as Minecraft does not support dynamic arrays out of the box. This means that every array element is wrapped in an object with a special accessor. The following array `[10, true,"Hello",[],{}]` is stored with this NBT: `[{_:10},{_:true},{_:"Hello"},{_:[]},{_:{}}]` and can be used like any ordinary array with an index and the special accessor: `my_array[2]._ #"Hello"`
6. **Typed Arrays** `[]` are stored as usual arrays in storages natively.
7. **Dynamic Objects** `{}` are stored in storages with special formatting similar to dynamic arrays because the entries cannot be accessed via index, unlike arrays. This

means that all keys of the object are indexed inside a wrapper object in an iterable array. The following object `{test:10,hello:"World",arr:[2,3,4]}` is stored with this NBT: `{_keys:["test","hello","arr"],_data:{test:10,hello:"World",arr:[2,3,4]}}` and can be iterated by iterating over the `_keys` array and retrieving the actual value of the object using a macro: `obj._data.$(_key)`.

8. **Typed Objects** `{}` are stored in the exact same way as dynamic objects, but enforce a strict type policy for the object's values to ensure type safety when accessing values.

3.2 Code Transpilation

Conditions: In order to evaluate conditions, we simply have to make sure we early return, if the evaluation is false:

```
# condition evaluation (whatever is necessary to calculate)
execute if score _out amethyst matches 0 run return fail
```

Branching: Logical control flow is handled in a special way to allow for branching with `if` and `else` - something that is hardly possible in regular MCF.

Language Reference

```
if (age >= 12) {
    # do some stuff
} else {
    debug "Must be at least 12 years old"
}
```

calling.mcfunction

```
execute if function if_func run return 1
execute if score _brk amethyst matches 1 run return fail
execute if score _ctn amethyst matches 1 run return fail
```

if.mcfunction

```
# evaluate condition
# evaluate if branch
execute if score _out amethyst matches 1 run return run
    function then_branch
# evaluate else branch
return run function else_branch
```

Loops: In MCF, loops can always defined through while true loops with a breaking condition. This means that all types of loops (for, while, ...) can be rewritten to the form of a while-true loop.

Language Reference

```
for (var x=0; x<10; x++) {
    # do some stuff
}
```

calling.mcfunction

```
execute if function loop run return 1
```

loop.mcfunction

```
# evaluate condition
# evaluate body
```

```
execute if score _brk amethyst matches 1 run return fail
scoreboard players set _ctn amethyst 0
function loop
```

Callable Functions: Normally, functions in MCF can only be called without parameters. Amethyst adds this functionality by pushing arguments onto a stack.

Language Reference

```
function my_func(arg1, arg2, arg3) {
    debug arg1 + ", " + arg2 + ", " + arg3;
}

[on_initialize]
function main() {
    my_func("Hello World", 10, {x: true}); # Hello World, 10, {
        x: true}
}
```

calling.mcfunction

```
# foreach all parameters:
# evaluate parameter
data modify storage amethyst:internal _argv append from storage
    amethyst:internal _out

function my_func

# for _ in 0..parameter_count
data remove storage amethyst:internal _argv[0]
```

my_func.mcfunction

```
# do some stuff and use parameters with _argv[-<n+1>]
```

3.3 More on Control Flow

As maybe already noticed, Amethyst internally heavily relies on the `return` command to control the logical flow through the program. However, the following concepts have to be kept in mind to understand what's actually happening:

1. `return` is used to completely return to a functions entry point, including breaking out of every `if`, `for` or whatsoever. This call is denoted in the source code using the MCF equivalent `return 1` which basically returns from the current function with a successful value 1.
2. `break` is used to only escape the current control flow like `for` or `while`-loops. To distinguish this behaviour from the full return, `return fail` is used instead with an additional `_brk` flag that is set to 1.
3. `continue` is used to escape the current control flow up until the first loop by passing `return fail` up the call-tree with an additional `_ctn` flag that is set to 1. In the loop, the `_ctn` flag will be acknowledged, reset to 0, and continues execution like usual.

The intended behaviour can be best explained by looking at the actual generated datapack code for the following Amethyst code:

```
for (var x = 0; x < 10; x = x+1) {
    if (x > 5) {
        return; # or break or continue (check the difference);
    }
}
```