Glossario

Datapack: gruppo di funzioni e altri file, paragonabile al "progetto" vero e proprio.
.mcfunction: formato di file, definibile come funzione che contiene comandi e commenti. Vengono messe nella cartella `function` dei datapack
Comando: qualsiasi riga di codice in un file mcfunction (ad esempio `scoreboard`, `data`, `function`.
Entità: Qualsiasi cosa dinamica e interattiva che esiste nel mondo ma che non appartiene alla griglia dei blocchi. Le entità includono giocatori, mostri, proiettili e altro.
Funzione: gruppo di comandi scritti in un file `.mcfunction`
NBT data: rappresentazione dati paragonabili a JSON, tuttavia la chiave non è circondata da `"`. Tutte le entità, oggetti e certi blocchi la possiedono.
Macro: feature relativamente nuova che consente di inserire valori in funzioni da una sorgente NBT (con casting implicito a stringa). Un comando che contiene una macro inizia sempre con `$`

```
test_macro.mcfunction:
    $say $(parameter) // $(<paramatro>)
    $say $(integer).$(decimal) //esempio di concatenazione di stringhe
macro_call.mcfunction:
    function my_pack:test_macro {parameter:"ciao",integer:1,decimal:6}

test_macro("ciao",1,0)

void test_macro(parameter,integer,decimal){

}

// output:
// ciao
// 1.5
```

Selettori: usati per selezionare entità applicando controlli quali distanza dal contesto di esecuzione, valori delle loro scoreboard, NBT e altro. Ad esempio `say @e[nbt={Health:20f}]` seleziona tutte le entità con 20 punti vita (`@e` seleziona tutte le entità).
Contesto di esecuzione: l'entità e posizione su cui sta venendo eseguito il comando. L'entità che sta eseguendo il comando è selezionabile tramite `@s` (at self). Il contesto di esecuzione rimane lo stesso nelle chiamate di funzioni seguenti a meno che non venga esplicitamente cambiato.
Command stack: gli sviluppatori hanno implementato i comandi per essere inviati al gioco uno alla volta (non esiste multithreading), prendendo il primo in cima ad uno stack.
UUID: ogni entità ha un Universally Unique Identifier è un long di 128 bit usato per distinguere le entità. Può essere rappresentato con

- esadecimale: `f81d4fae-7dec-11d0-a765-00a0c91e6bf6` si possono selezionare entità con un certo UUID solo se si ha il loro formato esadecimale.
- int-array: `[I;-132296786,2112623056,-1486552928,-920753162]`. Memorizzato nell'nbt delle varie entità

I comandi più importanti

- `execute`: consente di eseguire comandi se sono soddisfatte certe condizioni e di spostare il contesto di esecuzione da un entità a un altra o da una coordinata a un altra. `execute as @a at @s run setblock ~ ~-1 ~ stone` esegue i seguenti passi
  - seleziona tutti i giocatori e per ognuno di essi
  - imposta il giocatore come entità che sta eseguendo il comando (`as @a`)
  - sposta il luogo di esecuzione dei comandi a dove si trova il giocatore `at @s`
  - relativamente a quella posizione, al blocco sotto metti un blocco di pietra (`setblock ~ ~-1 ~ stone`)
- `say`: stampa stringhe su console
- `tellraw`: stampa strutture più complesse e ha una struttura NBT (simile a JSON)
- `function namespace:path/to/function`: chiama ed esegue una funzione
- `scoreboard`: esegue operazioni tra interi https://minecraft.wiki/w/Scoreboard#Score_operations
- `data`: legge e modifica dati NBT da qualsiasi sorgente (entità, blocchi, storage)
- `schedule`: esegue una funzione dopo un intervallo di tempo resettando il contesto di esecuzione.

Necessità di definire `scoreboard` prima di poterle usare per fare operazioni matematiche

Non si può fare `scoreboard players operation a math += b math` senza aver prima dichiarato la scoreboard `math` con `scoreboard objectives add math dummy`. `dummy` è un tipo di `objective` speciale che non viene influenzato da qualsiasi evento di gioco che non sia il comando `scoreboard` ed è sempre usato quando si vogliono fare operazioni matematiche.

Necessità di utilizzare namespace per evitare conflitti tra progetti diversi

Nell'esempio precedente, se due progetti diversi usano il nome `math` per le loro scoreboard, possono esserci dei conflitti se da un lato viene fatto `scoreboard players set temp dummy 1` e dall'altro `scoreboard players set temp dummy 2`. Per evitare questo problema si usano namespace per ogni progetto: `scoreboard players set temp foo.dummy 1` e `scoreboard players set temp bar.dummy 2`.
Questo problema non è limitato a scoreboard ma si applica anche a

- Tags: stringhe memorizzate in un array, usate per identificare/filtrare delle entità specifiche. Ad esempio, se aggiungo uno zombie che è molto lento ma ha molta vita, non voglio che questo venga selezionato dalle funzioni di un altro datapack che rende tutti gli zombie più veloci. Per questo il mio zombie avrà `Tags:[foo.custom_zombie]` e l'altro avrà `Tags:[bar.custom_zombie]`.
- Storage: i data storage sono spazi di memoria persistente che possono essere modificati o letti. Permettono di memorizzare dati non collegati a entità, blocchi o oggetti. Anche questi devono avere namespace perché se dichiaro uno storage che memorizza i dati di un oggetto: `data modify storage my_storage item set value {id:"diamond_sword",count:1}`. Se un altro datapack lo modifica con `data modify storage my_storage item set value {id:"dirt",count:64}`, e in seguito io volevo dare l'oggetto memorizzato su storage a un giocatore, esso avrà una 64 blocchi di terra piuttosto che una spada in diamante. Dunque per evitare questi conflitti entrambi gli sviluppatori usano namespace diversi:

```
data modify storage foo:my_storage item set value
{id:"diamond_sword",count:1}

data modify storage my_storage item set value {id:"dirt",count:64}
```

Utilizzo di simboli illegali per non impostare scoreboard con giocatori

Dato che le scoreboard sono state pensate per assegnare punteggi a giocatori reali, la sintassi è
`scoreboard players set <selector> <objetive> <value>`. il selector accetta nomi di giocatori, quindi ci si potrebbe trovare nel caso in cui un valore è assegnato a un giocatore. Il 99% delle volte questo non crea problemi, tuttavia per essere sicuri di non selezionare giocatori si inseriscono nei selettori simboli che non possono essere messi nel proprio username quando si crea un account.
Lo standard è `$` per le variabili e `#` per le costanti. Ad esempio `scoreboard players set #100 math 100`

Necessità di utilizzare costanti già dichiarate per fare operazioni diverse da somma e sottrazione

Riprendendo il paragrafo precedente, se si vuole moltiplicare o dividere un valore per un numero costante, bisogna aver già dichiarato quella costante in precedenza.
La somma e sottrazione di valori costanti è supportata con
`scoreboard players <add/remove> <selector> <scoreboard> <value>`
Ma non si può fare lo stesso con moltiplicazioni e divisioni. `x*=5` si deve scrivere come

```
scoreboard players set #5 math 5
scoreboard players operation $x math *= #5 math
```

la dichiarazione delle costanti va fatta solo una volta dato che in teoria non verranno mai modificate.

Non si può fare `c = a+b` lasciando `a` e `b` invariati (con una sola operazione)

Le operazioni di somma, moltiplicazione e divisione possono essere utilizzate solo come `a+=b` (per questo c'è `=` dopo gli operatori `+`, `-`, `/` e `*`. `c = a+b` si può scrivere nel seguente modo:

```
scoreboard players operation $c math = $a math
scoreboard players operation $c math += $b math
```

che equivale a

```
int c = a
c+=b
```

e per ogni operazione in più va aggiunta un altra riga

Operazioni limitate a int a 32 bit

Il dominio delle operazioni tra scoreboard è $\mathbb{Z} \cap [-2^{31}, 2^{31} - 1]$. Divisioni che avrebbero risultati decimali vengono approssimate all'intero più vicino. Ad esempio se si vuole calcolare il 5% di 40, bisogna stare attenti all'ordine delle operazioni:
(ometto la dichiarazione delle costanti `#100` e `#5`)

```
scoreboard players operation set $temp math 40
scoreboard players operaton $temp math /= #100 math
scoreboard players operation $temp math *= #5 math
```

$= 0$ dato che $\frac{40}{100} = 0$, mentre

```
scoreboard players operation set $temp math 40
scoreboard players operation $temp math *= #5 math
scoreboard players operaton $temp math /= #100 math
```

$= 2$, dato che $40 \times 5 > 100$, quindi dividendolo per 100 non verrà approssimato a 0.
Per evitare questi problemi di perdita di precisione in genere si moltiplica e poi divide per 10^n.

Si moltiplica il valore per 100 o 1000 a seconda della precisione richiesta, si fanno le operazioni, e poi lo si divide per 100 o 1000 memorizzando il valore in uno storage (dato che sono in grado di contenere double, float e long.
Per esempio se voglio indicare la percentuale di mana di un giocatore mostrando anche le cifre decimali

```
// ipotizzo di aver 41 mana e 53 max_mana
scoreboard players operation $mana_perc math = @s mana
scoreboard players operation $mana_perc math *= #10000 math
scoreboard players operation $mana_perc math /= @s max_mana
execute store result storage my_storage perc_mana int 0.01 run scoreboard
players get $mana_perc math
tellraw @s ["Max Mana",{"nbt":"perc_mana","storage":"my_storage"}] //
comando per stampare al giocatore
```

Tuttavia questo metodo può essere problematico se scalando, il valore è maggiore di $2^{31}-1$. Se $\text{mana} = 4100000$ e $\max \text{mana} = 5100000$,
$\text{mana\_perc} = -3.82$.

Assenza di C-style code blocks

Non si può raggruppare codice in blocchi, quindi se voglio eseguire comandi solo se una certa condizione è vera devo creare una nuova funzione. Questa è una tecnica comune in programmazione quando le righe di codice magari sono numerose, o per motivi di chiarezza. Tuttavia in mcfunction

- le funzioni sono in un altro file. Quindi bisogna passare da un tab all'altro per vedere cosa sta facendo la funzione che si sta chiamando
- queste funzioni possono avere anche solo due comandi. Se per esempio voglio che tutti le entità che hanno preso danno ricevano forza e un oggetto posso scrivere:

```
execute as @e if entity @s[nbt={HurtTime:10s}] run effect give @s strength
execute as @e if entity @s[nbt={HurtTime:10s}] run item replace entity @s
weapon.mainhand stone_sword
```

Il problema è che accedere ai dati nbt delle entità è un processo costoso dato che deve essere tutta serializzata, quindi per migliorare la performance si in genere si scrive:

```
execute as @e if entity @s[nbt={HurtTime:10s}] run function my_pack:on_hurt

// on_hurt.mcfunction:
effect give @s strength
item replace entity @s weapon.mainhand stone_sword
```

Per fare questo bisogna creare un nuovo file, e considerando quanto spesso si utilizzano if nella programmazione in generale, si può ipotizzare quanto possa diventare noioso, anche senza considerare la scomodità di passare da un tab a un altro per leggere il codice.
Sarebbe ideale poter scrivere

```
execute as @e if entity @s[nbt={HurtTime:10s}] run {
    effect give @s strength
    item replace entity @s weapon.mainhand stone_sword
}
```

ma questo non è possibile.
Un altro caso in cui sarebbe comodo essere in grado di raggruppare comandi in blocchi senza dover creare un nuovo file è scrivere `if...else if ...else` o `switch` statements
Se voglio stampare chi ha vinto, controllando un valore intero, si può fare usando il comando `return` per interrompere l'esecuzione di comandi all'interno di una funzione.
Se vogliamo scrivere il seguente codice in .mcfunction

```
if(team==1) print("blue won")
else if(team==2) print("red won")
else if(team==3) print("green won")
else print("no one won")
```

si può scrivere:

```
execute if score $team var matches 1 run return run say blue won
// se il flusso di esecuzione raggiunge questo punto vuol dire che $team !=
1
execute if score $team var matches 2 run return run say red won
// se il flusso di esecuzione raggiunge questo punto vuol dire che $team !=
1 e 2
execute if score $team var matches 3 run return run say green won
// se il flusso di esecuzione raggiunge questo punto vuol dire che nessun
team ha vinto
say no one won
```

considerando sempre che se si vogliono eseguire più comandi bisogna creare una nuova funzione, ovvero un nuovo file per ogni caso, anche perché scrivendo

```
execute if score $team var matches 1 run return run say blue won
execute if score $team var matches 1 run return run fill 0 0 0 0 10 0
blue_wool // comando per riempire i blocchi tra due coordinate con "lana
blu"
```

il secondo comando (e tutti quelli dopo) non viene eseguito per via del return del comando precedente.

Un metodo alternativo a controllare ogni valore (evitare hardcoding) consiste nell'usare macro per controllare in modo dinamico il valore, e chiamare una funzione che ha come nome o contiene nel nome quel valore.

```
// if(team<1||team>3) team = -1
// si assegna un valore di default/fallback se il valore non è tra quelli che vogliamo
execute unless score $team var matches 1..3 run scoreboard players set $team var -1
execute store result storage my_storage team.value int 1 run scoreboard players get $team var
function mypack:check_value with storage my_storage team

// check_value.mcfunction:
$execute if score $team var matches $(value) run function mypack:team$(value)
```

poi si dovranno fare 4 funzioni chiamate `team1.mcfunction`, `team2.mcfunction`, `team3.mcfunction`, `team-1.mcfunction`. Questo non è né chiaro ne rapido da scrivere se i casi sono molti.

Assenza di un (classico) array indexing

Riprendendo l'assenza di code blocks per il ciclo for citati nel paragrafo precedente, iterare su elementi di un array è fattibile in due modi

- combinazione di macro e ricorsione: complicato ma efficiente.

```
data modify storage my_storage food set value ["apple","chicken","bread"]
execute store result score $length var run data get storage my_storage food
execute if score $length var matches 1.. run function my_pack:iter {i:0}

// iter.mcfunction:
$tellraw @p {"nbt":"food[$(i)]","storage":"my_storage"}
$scoreboard players set $i var $(i)
scoreboard players add $i var 1
execute if score $i var >= $length run return fail // si esce dal ciclo se l'indice supera la lunghezza

exeute store result storage my_storage data.i int 1 run scoreboard players get $i var
function my_pack:iter storage my_storage data // si passa lo storage alla funzione da cui verrà presa la i
```

- copia e ricorsione: facile ma costoso, dato che si copia un intero array

```
data modify storage my_storage food set value ["apple","chicken","bread"]

data modify storage my_storage temp set from storage my_storage food //
copia dell'array da food a temp
function mypack:iter

// iter.mcfunction:
tellraw @p {"nbt":"temp[0]","storage":"my_storage"}
data remove storage my_storage temp[0]
if data storage my_storage temp[] run function mypack:iter // se ci sono
elementi in temp, ripeti
```

Assenza di logaritmi, esponenti, funzioni trigonometriche

Per ricreare queste funzioni si possono usare algoritmi che le approssimano basati su scoreboard

Radice quadrata

```
// approssimazione della radice quadrata dato input #x sqrt
execute store result score #t1 sqrt store result score #t2 sqrt store result
score #t3 sqrt run scoreboard players operation #y sqrt = #x sqrt
execute if score #x sqrt matches 0..1515359 run scoreboard players operation
#y sqrt /= #559 sqrt
execute if score #x sqrt matches 0..1515359 run scoreboard players add #y
sqrt 15
execute if score #x sqrt matches 1515360.. run scoreboard players operation
#y sqrt /= #32768 sqrt
execute if score #x sqrt matches 1515360.. run scoreboard players add #y
sqrt 2456
scoreboard players operation #t1 sqrt /= #y sqrt
scoreboard players operation #y sqrt += #t1 sqrt
scoreboard players operation #y sqrt /= #2 sqrt
scoreboard players operation #t2 sqrt /= #y sqrt
scoreboard players operation #y sqrt += #t2 sqrt
scoreboard players operation #y sqrt /= #2 sqrt
scoreboard players operation #t3 sqrt /= #y sqrt
scoreboard players operation #y sqrt += #t3 sqrt
scoreboard players operation #y sqrt /= #2 sqrt
scoreboard players operation #x sqrt /= #y sqrt
execute if score #y sqrt > #x sqrt run scoreboard players remove #y sqrt 1
```

Un esempio di applicazione della radice quadrata è calcolare la distanza tra due entità, dato che la loro posizione è memorizzata in un array di double.

Seno

Oppure si usano *lookup table* e macro per ottenere il valore richiesto (esempio per $\sin(x)$ `$execute store result score $sine player_motion.internal.math run data get storage player_motion:sine arr[$(angle)]` (esempio preso da una libreria che applica un impulso di moto ad un giocatore in base a dove esso sta guardando).
Anche qua si possono usare algoritmi che approssimano:

```
// init
scoreboard players set #sign math -400
// setup input
// you can remove this first line if you are sure that your input is between
0.0 and 360.0
scoreboard players operation .in math %= #3600 const
execute if score .in math matches 1800.. run scoreboard players set #sign
math 400
execute store result score #temp math run scoreboard players operation .in
math %= #1800 const
// run
scoreboard players remove #temp math 1800
execute store result score .out math run scoreboard players operation #temp
math *= .in math
scoreboard players operation .out math *= #sign math
scoreboard players add #temp math 4050000
scoreboard players operation .out math /= #temp math
// you can remove this line if you dont want the extra 0.01 precision for
180.0-360.0 input range
execute if score #sign math matches 400 run scoreboard players add .out math
1
```

Coseno

Si usa l'approsimazione di Bhaskara

```
// Transform input
scoreboard players operation #output math = #input math
scoreboard players remove #output math 900
scoreboard players operation #output math %= #1800 math
scoreboard players remove #output math 900
// Compute Bhaskara's approximation

scoreboard players operation #output math *= #output math
scoreboard players operation #math_trigonometry_0 math = #output math
scoreboard players add #math_trigonometry_0 math 3240000
scoreboard players operation #math_trigonometry_0 math /= #1000 math
scoreboard players operation #output math *= #4 math
scoreboard players operation #output math *= #-1 math
scoreboard players add #output math 3240000
scoreboard players operation #output math /= #math_trigonometry_0 math
// Apply sign
```

```
scoreboard players operation #math_trigonometry_0 math = #input math
scoreboard players add #math_trigonometry_0 math 900
scoreboard players operation #math_trigonometry_0 math %= #3600 math
execute if score #math_trigonometry_0 math matches 1800.. run scoreboard
players operation #output math *= #-1 math
```

codice equivalente in java:

```
double bhaskaraCosine(double degrees) {
        double output = degrees % 1800;
        output -= 900;
        output %= 1800;
        output -= 900;

        double squared = output * output;
        double denominator = squared + 3240000;
        denominator /= 1000;
        output *= -4;
        output += 3240000;
        output /= denominator;

        double checkSign = degrees + 900;
        checkSign %= 3600;
        if (checkSign >= 1800) {
            output *= -1;
        }

        return output / 1000.0;
    }
```

Trucchi "non ufficiali" per casi specifici

Ci sono dei "trucchi" specifici a Minecraft che possono essere utilizzati per calcolare velocemente certe funzioni.

Seno

Viene impostata la posizione di un entità di utilità a $0.0, 0.0, 0.0$ la sua rotazione sull'asse delle x viene impostata a quella dell'angolo richiesto, e poi teletrasportata 1 blocco in avanti rispetto a dove sta guardando, la sua coordinata X allora sarà il seno dell angolo richiesto.

```
$execute rotated $(x) 0 positioned 0.0 0.0 0.0 positioned ^ ^ ^-1 as 91bb5-
0-0-0-ffff run return run function my_pack:sin

// sin.mcfunction:
```

```
tp @s ~ ~ ~
data modify storage my_pack:io out set from entity @s Pos[0]
```

Distanza euclidea ($\sqrt{x^2+y^2+z^2}$)

si sfruttano le proprietà di un entità in genere usata per display di oggetti che usa quaternion, che per motivi non del tutto conosciuti risolve questi input.

```
$data modify entity @s transformation set value
[$(x)f,0f,0f,0f,$(y)f,0f,0f,0f,$(z)f,0f,0f,0f,0f,0f,0f,1f]

tellraw @p {"nbt":"transformation.scale[0]","entity":"@s"} // contiene
l'output
```

Impostando uno dei tre valori a 0, si può calcolare $\sqrt{a^2+b^2}$.

Somma e sottrazione tra double

Si sfrutta il fatto che i mondi hanno altezza pressoché illimitata, e si sposta un entità da un'altezza all'altra, memorizzando l'altezza finale.

```
$execute positioned ~ $(x) ~ run tp 91bb5-0-0-0-ffff 29999999 ~$(y) 91665 //
seleziona entità con UUID 91bb5-0-0-0-ffff
// la si sposta dall'altezza x all'altezza y
data modify storage my_pack:io out set from entity 91bb5-0-0-0-ffff Pos[1]
```

Assenza di operazioni bitwise

Considerando che i colori sono memorizzati in formato decimale (`#FF0000=16712965`) se voglio mischiare due colori al posto di usare il bit shifting:

```
int blendColors(int oldColor, int newColor) {
    // valore rgb per ogni colore
    int oldRed = (oldColor >> 16) & 0xFF;
    int oldGreen = (oldColor >> 8) & 0xFF;
    int oldBlue = oldColor & 0xFF;

    int newRed = (newColor >> 16) & 0xFF;
    int newGreen = (newColor >> 8) & 0xFF;
    int newBlue = newColor & 0xFF;

    // calcolo il valore medio dei due
    int blendedRed = (oldRed + newRed) / 2;
    int blendedGreen = (oldGreen + newGreen) / 2;
    int blendedBlue = (oldBlue + newBlue) / 2;
```

```
    // combino per ottenere il colore finale
    return (blendedRed << 16) | (blendedGreen << 8) | blendedBlue;
}
```

si estrae ogni canale per ogni colore con una serie di divisioni e poi lo si ricompone moltiplicando

```
// estratto di codice da un vecchio progetto dove si potevano mettere
ingredienti in un calderone e il colore dell'acqua era dato dal colore dei
due ingredienti

// Desc: Blends the old and new color together

// Convert old color into RGB
execute store result score $old_color acbag.dummy run data get entity @s
ArmorItems[3].tag.display.color
scoreboard players operation $old_red acbag.dummy = $old_color acbag.dummy
scoreboard players operation $old_red acbag.dummy /= acbag.const.65536
acbag.dummy
scoreboard players operation $remove acbag.dummy = $old_red acbag.dummy
scoreboard players operation $remove acbag.dummy *= acbag.const.65536
acbag.dummy

scoreboard players operation $old_color acbag.dummy -= $remove acbag.dummy
scoreboard players operation $old_green acbag.dummy = $old_color acbag.dummy
scoreboard players operation $old_green acbag.dummy /= acbag.const.256
acbag.dummy
scoreboard players operation $remove acbag.dummy = $old_green acbag.dummy
scoreboard players operation $remove acbag.dummy *= acbag.const.256
acbag.dummy

scoreboard players operation $old_color acbag.dummy -= $remove acbag.dummy
scoreboard players operation $old_blue acbag.dummy = $old_color acbag.dummy

// Convert new color into RGB
execute store result score $new_color acbag.dummy run data get storage
acbag:storage root.temp.cauldron.tempItem.tag.acbag.brew.color
scoreboard players operation $new_red acbag.dummy = $new_color acbag.dummy
scoreboard players operation $new_red acbag.dummy /= acbag.const.65536
acbag.dummy
scoreboard players operation $remove acbag.dummy = $new_red acbag.dummy
scoreboard players operation $remove acbag.dummy *= acbag.const.65536
acbag.dummy

scoreboard players operation $new_color acbag.dummy -= $remove acbag.dummy
scoreboard players operation $new_green acbag.dummy = $new_color acbag.dummy
scoreboard players operation $new_green acbag.dummy /= acbag.const.256
acbag.dummy
scoreboard players operation $remove acbag.dummy = $new_green acbag.dummy
```

```
scoreboard players operation $remove acbag.dummy *= acbag.const.256
acbag.dummy

scoreboard players operation $new_color acbag.dummy -= $remove acbag.dummy
scoreboard players operation $new_blue acbag.dummy = $new_color acbag.dummy

// calculate avarage
scoreboard players operation $new_red acbag.dummy += $old_red acbag.dummy
scoreboard players operation $new_green acbag.dummy += $old_green
acbag.dummy
scoreboard players operation $new_blue acbag.dummy += $old_blue acbag.dummy

scoreboard players operation $new_red acbag.dummy /= acbag.const.2
acbag.dummy
scoreboard players operation $new_green acbag.dummy /= acbag.const.2
acbag.dummy
scoreboard players operation $new_blue acbag.dummy /= acbag.const.2
acbag.dummy

// Convert back into decimal value
scoreboard players operation $new_red acbag.dummy *= acbag.const.65536
acbag.dummy
scoreboard players operation $new_green acbag.dummy *= acbag.const.256
acbag.dummy
scoreboard players operation $new_color acbag.dummy = $new_blue acbag.dummy
scoreboard players operation $new_color acbag.dummy += $new_green
acbag.dummy
scoreboard players operation $new_color acbag.dummy += $new_red acbag.dummy

// Insert value into item
execute store result entity @s ArmorItems[3].tag.display.color int 1 run
scoreboard players get $new_color acbag.dummy
```

Dividere il colore per $2^{16} = 65536$ equivale a spostare i bit a destra di 16 spazi per rendere il canale leggibile. Il processo opposto viene usato per ricomporre il colore finale dati i valori medi di R,G e B.

Non si può (sempre) usare JSON in file .mcfunction

Minecraft usa file JSON per dichiarare obiettivi (advancement), ricette (recipes), bottini dei bauli (loot tables) e altro. Scriverli non è un problema dato che ci sono generatori, e alcuni di questi possono essere scritti direttamente nel nel file .mcfunction senza creare un file .json a cui fare riferimento, ma non tutti. Per esempio se sto programmando una spada e voglio aggiungere una ricetta per costruirla, devo andare nella cartella specifica delle ricette e aggiungerla lì, poi se voglio fare un obiettivo che da premi quando la ottengo dovrò andare nella cartella degli obiettivi e creare un altro file JSON, e infine tornare alla mia cartella con il codice per la spada. Non è un problema grandissimo, ma considerando che è stata mostrata la comodità di poter scrivere più funzioni in un unico file, si può estendere questo concetto e

magari definire anche queste strutture json che di base richiederebbero un loro file nel medesimo file che già contiene i comandi per la spada.

```
package mysword

// my_sword.mcf (una ipotetica implementazione)
advancement detect_hit = {...}
recipe my_sword = {...}

function on_hit = {
    say you hit a monster
    if(...){
        say special hit
        effect give @s strength
    }
}
```

Avere tutti i file utilizzati per la stessa feature (in questo caso la spada) in un unico punto è molto più comodo di dover creare tutti questi file e cartelle e navigare tra gli uni e li altri

```
my_pack
├── function
│   └── my_sword
│       ├── on_hit.mcfunction
│       └── special_hit.mcfunction
├── recipe
│   └── my_sword.json
└── advancement
    └── detect_hit.json
```

Quaternion Math

Operazioni sui quaternioni sono richieste per trasformazioni complesse nello spazio tridimensionale, ad esempio ruotare un oggetto attorno al suo centro.

```
// Store quat rotation in storage
execute store result storage games:10 quat.w double 0.005 run scoreboard
players get @s 10quatw
execute store result storage games:10 quat.x double 0.005 run scoreboard
players get @s 10quatx
execute store result storage games:10 quat.y double 0.005 run scoreboard
players get @s 10quaty
execute store result storage games:10 quat.z double 0.005 run scoreboard
players get @s 10quatz
```

```
// Store current quaternion rotation in storage
execute store result storage general:math base.x double 0.0001 run data get
entity @s transformation.left_rotation[0] 10000
execute store result storage general:math base.y double 0.0001 run data get
entity @s transformation.left_rotation[1] 10000
execute store result storage general:math base.z double 0.0001 run data get
entity @s transformation.left_rotation[2] 10000
execute store result storage general:math base.w double 0.0001 run data get
entity @s transformation.left_rotation[3] 10000

// Set storage for old and new quaternion
data modify storage general:math quata set from storage games:10 quat
data modify storage general:math quatb set from storage general:math base

// Multiply quaternions
execute store result score #input_quaternion_1_r math run data get storage
general:math quata.w 1000
execute store result score #input_quaternion_1_i math run data get storage
general:math quata.x 1000
execute store result score #input_quaternion_1_j math run data get storage
general:math quata.y 1000
execute store result score #input_quaternion_1_k math run data get storage
general:math quata.z 1000
execute store result score #input_quaternion_2_r math run data get storage
general:math quatb.w 1000
execute store result score #input_quaternion_2_i math run data get storage
general:math quatb.x 1000
execute store result score #input_quaternion_2_j math run data get storage
general:math quatb.y 1000
execute store result score #input_quaternion_2_k math run data get storage
general:math quatb.z 1000

function general:math/quat/multiply2

execute store result storage games:10 quat.w double 0.001 run scoreboard
players get #output_quaternion_r math
execute store result storage games:10 quat.x double 0.001 run scoreboard
players get #output_quaternion_i math
execute store result storage games:10 quat.y double 0.001 run scoreboard
players get #output_quaternion_j math
execute store result storage games:10 quat.z double 0.001 run scoreboard
players get #output_quaternion_k math

function games:10/spin_can with storage games:10 quat // applico le nuove
trasformazioni a un barattolo quando colpito da un proiettile
```

```
// multiply2.mcfunction:
# Multiply quaternions
```

```
scoreboard players operation #math_quaternion_r math = #input_quaternion_1_r
math
scoreboard players operation #math_quaternion_r math *=
#input_quaternion_2_r math
scoreboard players operation #math_quaternion_i math = #input_quaternion_1_i
math
scoreboard players operation #math_quaternion_i math *=
#input_quaternion_2_i math
scoreboard players operation #math_quaternion_j math = #input_quaternion_1_j
math
scoreboard players operation #math_quaternion_j math *=
#input_quaternion_2_j math
scoreboard players operation #math_quaternion_k math = #input_quaternion_1_k
math
scoreboard players operation #math_quaternion_k math *=
#input_quaternion_2_k math

scoreboard players operation #output_quaternion_r math = #math_quaternion_r
math
scoreboard players operation #output_quaternion_r math -= #math_quaternion_i
math
scoreboard players operation #output_quaternion_r math -= #math_quaternion_j
math
scoreboard players operation #output_quaternion_r math -= #math_quaternion_k
math
scoreboard players operation #output_quaternion_r math /= #1000 math


scoreboard players operation #math_quaternion_r math = #input_quaternion_1_r
math
scoreboard players operation #math_quaternion_r math *=
#input_quaternion_2_i math
scoreboard players operation #math_quaternion_i math = #input_quaternion_1_i
math
scoreboard players operation #math_quaternion_i math *=
#input_quaternion_2_r math
scoreboard players operation #math_quaternion_j math = #input_quaternion_1_j
math
scoreboard players operation #math_quaternion_j math *=
#input_quaternion_2_k math
scoreboard players operation #math_quaternion_k math = #input_quaternion_1_k
math
scoreboard players operation #math_quaternion_k math *=
#input_quaternion_2_j math

scoreboard players operation #output_quaternion_i math = #math_quaternion_r
math
scoreboard players operation #output_quaternion_i math += #math_quaternion_i
math
scoreboard players operation #output_quaternion_i math += #math_quaternion_j
```

```
math
scoreboard players operation #output_quaternion_i math -= #math_quaternion_k
math
scoreboard players operation #output_quaternion_i math /= #1000 math


scoreboard players operation #math_quaternion_r math = #input_quaternion_1_r
math
scoreboard players operation #math_quaternion_r math *=
#input_quaternion_2_j math
scoreboard players operation #math_quaternion_i math = #input_quaternion_1_i
math
scoreboard players operation #math_quaternion_i math *=
#input_quaternion_2_k math
scoreboard players operation #math_quaternion_j math = #input_quaternion_1_j
math
scoreboard players operation #math_quaternion_j math *=
#input_quaternion_2_r math
scoreboard players operation #math_quaternion_k math = #input_quaternion_1_k
math
scoreboard players operation #math_quaternion_k math *=
#input_quaternion_2_i math

scoreboard players operation #output_quaternion_j math = #math_quaternion_r
math
scoreboard players operation #output_quaternion_j math -= #math_quaternion_i
math
scoreboard players operation #output_quaternion_j math += #math_quaternion_j
math
scoreboard players operation #output_quaternion_j math += #math_quaternion_k
math
scoreboard players operation #output_quaternion_j math /= #1000 math


scoreboard players operation #math_quaternion_r math = #input_quaternion_1_r
math
scoreboard players operation #math_quaternion_r math *=
#input_quaternion_2_k math
scoreboard players operation #math_quaternion_i math = #input_quaternion_1_i
math
scoreboard players operation #math_quaternion_i math *=
#input_quaternion_2_j math
scoreboard players operation #math_quaternion_j math = #input_quaternion_1_j
math
scoreboard players operation #math_quaternion_j math *=
#input_quaternion_2_i math
scoreboard players operation #math_quaternion_k math = #input_quaternion_1_k
math
scoreboard players operation #math_quaternion_k math *=
#input_quaternion_2_r math
```

```
scoreboard players operation #output_quaternion_k math = #math_quaternion_r
math
scoreboard players operation #output_quaternion_k math += #math_quaternion_i
math
scoreboard players operation #output_quaternion_k math -= #math_quaternion_j
math
scoreboard players operation #output_quaternion_k math += #math_quaternion_k
math
scoreboard players operation #output_quaternion_k math /= #1000 math
```

Trascurando il tempo impiegato a scrivere ogni singola operazione, bisogna stare attenti a scrivere le operazioni nell'ordine giusto (ad esempio prima si fanno quelle tra parentesi, poi le moltiplicazioni, ecc).
In un linguaggio vero questa moltiplicazione può essere scritta nel seguente modo:

```
public class Quaternion {
    double w, x, y, z;

    public Quaternion(double w, double x, double y, double z) {
        this.w = w;
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public Quaternion multiply(Quaternion q) {
        double newW = this.w * q.w - this.x * q.x - this.y * q.y - this.z *
q.z;
        double newX = this.w * q.x + this.x * q.w + this.y * q.z - this.z *
q.y;
        double newY = this.w * q.y - this.x * q.z + this.y * q.w + this.z *
q.x;
        double newZ = this.w * q.z + this.x * q.y - this.y * q.x + this.z *
q.w;

        return new Quaternion(newW, newX, newY, newZ);
    }
}
```

Convertire UUID da int array a hex

```
data modify storage cgn:macro root.data set value
{0:0,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,a:0,b:0,c:0,d:0,e:0,f:0}

data modify storage cgn:macro root.in set from entity @s UUID
execute store result score $uuid0 cgn.dummy run data get storage cgn:macro
```

```
root.in[0]
execute store result score $uuid1 cgn.dummy run data get storage cgn:macro
root.in[1]
execute store result score $uuid2 cgn.dummy run data get storage cgn:macro
root.in[2]
execute store result score $uuid3 cgn.dummy run data get storage cgn:macro
root.in[3]
// scomposizione dei 4 int in 16 byte da 0 a f
execute store result score 0= cgn.dummy run scoreboard players operation 1=
cgn.dummy = $uuid0 cgn.dummy
execute store result storage cgn:macro root.data.0 int 1 run scoreboard
players operation 0= cgn.dummy %= #256 cgn.dummy
execute store result score 2= cgn.dummy run scoreboard players operation 1=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.1 int 1 run scoreboard
players operation 1= cgn.dummy %= #256 cgn.dummy
execute store result score 3= cgn.dummy run scoreboard players operation 2=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.2 int 1 run scoreboard
players operation 2= cgn.dummy %= #256 cgn.dummy
execute store result storage cgn:macro root.data.3 int 1 run scoreboard
players operation 3= cgn.dummy /= #256 cgn.dummy

execute store result score 0= cgn.dummy run scoreboard players operation 1=
cgn.dummy = $uuid1 cgn.dummy
execute store result storage cgn:macro root.data.4 int 1 run scoreboard
players operation 0= cgn.dummy %= #256 cgn.dummy
execute store result score 2= cgn.dummy run scoreboard players operation 1=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.5 int 1 run scoreboard
players operation 1= cgn.dummy %= #256 cgn.dummy
execute store result score 3= cgn.dummy run scoreboard players operation 2=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.6 int 1 run scoreboard
players operation 2= cgn.dummy %= #256 cgn.dummy
execute store result storage cgn:macro root.data.7 int 1 run scoreboard
players operation 3= cgn.dummy /= #256 cgn.dummy

execute store result score 0= cgn.dummy run scoreboard players operation 1=
cgn.dummy = $uuid2 cgn.dummy
execute store result storage cgn:macro root.data.8 int 1 run scoreboard
players operation 0= cgn.dummy %= #256 cgn.dummy
execute store result score 2= cgn.dummy run scoreboard players operation 1=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.9 int 1 run scoreboard
players operation 1= cgn.dummy %= #256 cgn.dummy
execute store result score 3= cgn.dummy run scoreboard players operation 2=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.a int 1 run scoreboard
players operation 2= cgn.dummy %= #256 cgn.dummy
```

```
execute store result storage cgn:macro root.data.b int 1 run scoreboard
players operation 3= cgn.dummy /= #256 cgn.dummy

execute store result score 0= cgn.dummy run scoreboard players operation 1=
cgn.dummy = $uuid3 cgn.dummy
execute store result storage cgn:macro root.data.c int 1 run scoreboard
players operation 0= cgn.dummy %= #256 cgn.dummy
execute store result score 2= cgn.dummy run scoreboard players operation 1=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.d int 1 run scoreboard
players operation 1= cgn.dummy %= #256 cgn.dummy
execute store result score 3= cgn.dummy run scoreboard players operation 2=
cgn.dummy /= #256 cgn.dummy
execute store result storage cgn:macro root.data.e int 1 run scoreboard
players operation 2= cgn.dummy %= #256 cgn.dummy
execute store result storage cgn:macro root.data.f int 1 run scoreboard
players operation 3= cgn.dummy /= #256 cgn.dummy

function cgn:technical/string_uuid/get_hexes with storage cgn:macro
root.data
function cgn:technical/string_uuid/concat_uuid with storage cgn:macro
root.data
```

```
// get_hexes.mcfunction:
// usa una lookup_table
//
https://github.com/asdru22/CognitionDev/blob/main/datapack/data/cgn/function
/technical/load.mcfunction#L43
$data modify storage cgn:macro root.data.0 set from storage cgn:storage
root.hex_chars[$(0)]
$data modify storage cgn:macro root.data.1 set from storage cgn:storage
root.hex_chars[$(1)]
$data modify storage cgn:macro root.data.2 set from storage cgn:storage
root.hex_chars[$(2)]
$data modify storage cgn:macro root.data.3 set from storage cgn:storage
root.hex_chars[$(3)]
$data modify storage cgn:macro root.data.4 set from storage cgn:storage
root.hex_chars[$(4)]
$data modify storage cgn:macro root.data.5 set from storage cgn:storage
root.hex_chars[$(5)]
$data modify storage cgn:macro root.data.6 set from storage cgn:storage
root.hex_chars[$(6)]
$data modify storage cgn:macro root.data.7 set from storage cgn:storage
root.hex_chars[$(7)]
$data modify storage cgn:macro root.data.8 set from storage cgn:storage
root.hex_chars[$(8)]
$data modify storage cgn:macro root.data.9 set from storage cgn:storage
root.hex_chars[$(9)]
$data modify storage cgn:macro root.data.a set from storage cgn:storage
```

```
root.hex_chars[$(a)]
$data modify storage cgn:macro root.data.b set from storage cgn:storage
root.hex_chars[$(b)]
$data modify storage cgn:macro root.data.c set from storage cgn:storage
root.hex_chars[$(c)]
$data modify storage cgn:macro root.data.d set from storage cgn:storage
root.hex_chars[$(d)]
$data modify storage cgn:macro root.data.e set from storage cgn:storage
root.hex_chars[$(e)]
$data modify storage cgn:macro root.data.f set from storage cgn:storage
root.hex_chars[$(f)]
```

```
// concat_uuid.mcfunction
// concatenazione dei valori
$data modify storage cgn:storage root.temp.string_uuid set value
"$(3)$(2)$(1)$(0)-$(7)$(6)-$(5)$(4)-$(b)$(a)-$(9)$(8)$(f)$(e)$(d)$(c)"
```

Questa funzione mostra anche come sia possibile concatenare le stringhe (normalmente si scrive semplicemente `String pizza = "piz"+"za"`).
Questa conversione viene fatta spesso dato che se si vuole memorizzare un entità da un contesto precedente e poi usarla nel contesto di un'altra è più efficiente usare hex che un selettore che deve serializzare tutta l'nbt di un entità.

```
// l'entità che sta eseguendo il comando (@s) subirà 10 danno "inflitto"
dall'altra.
damage @s 10 by f81d4fae-7dec-11d0-a765-00a0c91e6bf6
// vs
damage @s 10 by @n[nbt={UUID:
[I;-132296786,2112623056,-1486552928,-920753162]}]
```

Critiche alla proposta di tesi

- https://github.com/gibbsly/gm: libreria che usa strucchi basati su Trucchi "non ufficiali" per casi specifici per calcolare le principali funzioni matematiche in maniera veloce, che funziona anche con double. Ha lo svantaggio di essere più costosa in termini di tempo, ma se usata in maniera parsimoniosa è quasi impercettibile.
- Snapshot del 26/02 che aggiorna la formattazione dei numeri in NBT. Potrebbe indicare a un potenziale aggiornamento sul modo di fare operazioni matematiche con i comandi.

  “Pasted image 20250228004249.png” could not be found.
- esistono altri già pre-compilatori che convertono un qualche linguaggio in .mcfunction: https://gist.github.com/Ellivers/db296c438f9f87bbf9c79d24f940fe03, il più popolare al momento è https://github.com/mcbeet/beet.