0. Contesto

Questi esempi sono presi dal progetto allegato che contiene il mio codice ideale per implementare un blocco, chiamato monolite lunare che ha un suo modello, texture, suoni e comportamenti in base al tempo del giorno. C'è anche un oggetto "stone glyph" (anch'esso con modello e texture) che interagisce con il blocco.

> Dato che mi ha chiesto come sarebbe lo sviluppo ideale di un progetto del genere, ci sono anche alcuni punti relativi alla gestione delle risorse (4,15), che non essendo strettamente collegati alla parte di codice non sono sicuro rientrino del tema di ricerca (o comunque sono secondari).
> Sono cosciente del fatto che le risorse in applicazioni java, o app mobili, siano in una cartella a parte con suoni, immagini traduzioni. Questo modello si ritrova anche in minecraft dove c'è una divisione tra la parte *data*, relativa alle modifiche dei comportamenti del gioco, e *assets*: contenente suoni, immagini, modelli 3d, traduzioni...
> Ho ripreso l'esempio passato dove ho mostrato la scomodità di trovarsi a lavorare con file di categorie diverse, la maggior parte dei quali in genere di poche righe, e aggiunto la parte relativa alle risorse per un oggetto estremamente banale `my_item`. Anche questi sono file json in genere di piccole dimensioni, oppure `.ogg` per i suoni o `.png` per le immagini.

```
my_project
├── datapack
│   └── data
│       └── namespace
│           ├── function
│           │   └── my_item
│           │       ├── some_code.mcfunction
│           │       └── other_code.mcfunction
│           └── recipe
│               └── my_item.json
└── resourcepack
    └── assets
        └── namespace
            ├── items
            │   └── my_item.json
            ├── models
            │   └── item
            │       └── my_item.json
            ├── texture
            │   └── item
```

```
│       └── my_item.png
└── lang
    └── en_us.json
```

> Nel video in allegato spero si noti la scomodità di dover lavorare con la parte di codice così "lontana" da quella delle risorse (in un ambiente di lavoro di un progetto abbastanza grande, ottimizzato il più possibile per ridurre il numero di sottocartelle, rispettando i vincoli di compilazione di Minecraft). In conclusione, secondo me sarebbe ideale concentrare il più possibile *data* e *assets* relativi a una certa feature il più possibile, similmente a quanto già detto all'ultimo incontro per i file quali `recipe`, `loot_table`, `advancement`...

> Un altra nota, non sono sicuro se sia meglio usare la notazione `.` simile a quella di java rispetto a `/`, già usata di default da mcfunction e molti altri generatori che creano file json.

1. Referenziazione a namespace

`*` corrisponde al namespace, in questo caso `haywire` seguito da `.` (carattere convenzionale separatore, potenzialmente da far specificare all'utente) per tags, scoreboard, bossbar, teams ("oggetti" di minecraft con scope globale, che possono creare conflitti se hanno lo stesso nome) e `:` per riferimenti a file esterni (json o funzioni, con l'eccezione dei modelli, di cui parlo più avanti) e storage (il compilatore deve essere in grado di capire se il `*` si sta riferendo a un file, storage o ad oggetti specifici di minecraft).
Potenzialmente se l'uso di `*` non ricade in questi due casi ma ad esempio viene usato come chiave di una struttura nbt `*:{id:"my_item"}` il compilatore potrebbe semplicemente sostituirlo con il namespace specificato nelle impostazioni del progetto.
Non deve essere per forza questo carattere, magari definibile dall'utente.

> Mi sono accorto tardi che ci sono casi in cui mcfunction usa `*` come parola chiave valida quindi forse è meglio trovare un altro carattere/altra soluzione

2. Package

Si potrebbe far specificare all'utente se preferiscono usare `.` o `/` per navigare dentro i package. Non sono sicuro sia necessario definire il package, dato che coincide con la struttura delle cartelle/file (almeno per come ho organizzato i file io).
Secondo me il concetto di package come scorciatoia per indicare le cartelle e sottocartelle in cui saranno generati i file è utile solo nel caso delle funzioni: ovvero, se specifico il package `my_package.item` e in quel package c'è un file che dichiara e definisce una funzione `my_function` e una ricetta `my_recipe`, la funzione verra compilata in `function/my_package/item/my_function`, mentre la ricetta in `recipe/my_recipe`. Questo perché statisticamente le ricette e altri file json sono utilizzati meno in proporzione alle funzioni. Inoltre, essendo meno ci sono meno necessità di organizzarle in più sottocartelle.

3. Gestione file json

La prassi per creare i file json è con https://misode.github.io/, non sono sicuro se sia necessario trovare un modo per migliorare la loro scrittura. Sia con mcfunction tradizionale e questo linguaggio ideale si fa copia-incolla dal sito.
classico:

```
{
  "pools": [
    {
      "rolls": 1,
      "entries": [
        {
          "type": "minecraft:loot_table",
          "functions": [
            {
              "function": "minecraft:set_count",
              "count": {
                "type": "minecraft:score",
                "target": {
                  "type": "minecraft:fixed",
                  "name": "#count"
                },
                "score": "haywire.dummy"
              }
            }
          ],
          "value": "*items/moonlit_monolith"
        }
      ]
    }
  ],
  "random_sequence": "*blocks/moonlit_monolith"
}
```

ideale (il namespace è stato sostituito da `*` descritto nel punto 1):

```
{
  "pools": [
    {
      "rolls": 1,
      "entries": [
        {
          "type": "minecraft:loot_table",
          "functions": [
            {
              "function": "minecraft:set_count",
              "count": {
```

```
                "type": "minecraft:score",
                "target": {
                  "type": "minecraft:fixed",
                  "name": "#count"
                },
                "score": "*dummy"
              }
            }
          ],
          "value": "*items/moonlit_monolith"
        }
      ]
    }
  ],
  "random_sequence": "*blocks/moonlit_monolith"
}
```

4. Modelli 3D, texture e suoni

In questa implementazione di progetto, dove i file relativi al datapack (funzioni, loot table, advancement,...) e quelli della cartella risorse (texture, suoni, modelli 3D) possono essere messi dove l'utente desidera e poi spetterà al compilatore stabilire dove poi verrano messi i file.
Bisogna anche considerare che i modelli 3d vengono creati molto facilmente o con https://misode.github.io/ o con un software specifico, blockbench (simile a blender), che applica le texture specificate agli oggetti 3d.
Ho lasciato quindi due esempi:

- il modello è semplice (composto da una sola texture, creato con https://misode.github.io/ dove la preview del modello finale non è importante). In questo caso il modello l ho dichiarato nel file .mcf assieme agli altri dati necessari, e la texture viene scelta con la feature del punto 15.
- il modello è complesso (caso del blocco, realizzato con blockbench): in questo caso a ogni modello corrisponde un file json, che può essere aperto da blockbench.

5. Scope dei nomi file

```
predicate value_check.is_night = { // predicates/value_check/night (non
fatto inline perché riutilizzato)
  "condition": "minecraft:time_check",
  "value": {
    "min": 13000,
    "max": 22999
  }
}
```

Se ho capito bene da quello che ci siamo detti, si può fare riferimento a un file del genere semplicemente prendendo l'ultimo valore del suo "path". In questo caso il nome del predicate è `night`. Si rischia di avere magari un predicate e una funzione con stesso nome. Suppongo spetti al compilatore scegliere il file giusto in base al contesto? altrimenti si può sollevare un eccezione che dice che i due file hanno lo stesso nome anche se in mcfunction normale non sarebbe un problema dato che predicate e function sono in cartelle diverse.

6. Scorciatoie scoreboard

Al posto di scrivere

```
scoreboard players set #8 haywire.dummy 8

scoreboard players operation #moon_phase haywire.dummy %= #8 haywire.dummy
```

si potrebbe scrivere `operation #moon_phase *dummy %= 8` (avrei voluto usare `op` ma c'è un altro comando con lo stesso nome che non ha nulla a che fare con le operazioni), dove il valore 8 associato a `#8` è impostato magari in un file specializzato per le costanti quando si compila il progetto.
Però, se si fanno operazioni di somma o sottrazione (`operation @s *dummy2 +=/-= 1`), si possono generare comandi del tipo `scoreboard players add/remove @s haywire.dummy2 1` che non richiedono l'assegnamento di una costante prima di essere usati (c'è il comando specifico solo per sommare/sottrare un valore costante). Di conseguenza si potrebbe anche implementare `operation @s *dummy2 ++` per incrementare/decrementare un valore di 1.

7. Scorciatoie execute

Si potrebbe omettere il comando `execute` e trattare i suoi sottocomandi come il comando vero e proprio. `execute` è il comando più potente, che ha dei sottocomandi quali `if` per verificare una certa condizione, `as` per spostare l'esecuzione dei comandi su un altra entità ecc. Al posto di scrivere `execute as @p at @s ...` si potrebbe semplicemente scrivere `as @p at @s ...`, dato che non esistono comandi con nomi uguali a quelli dei sottocomandi di execute.

8. Aggiungere comandi in una funzione da un altro file

Ovviamente questo può essere fatto solo in casi dove l'ordine di esecuzione dei comandi non importa a discrezione dell'utente. Ad esempio

```
append(*block.timers.10_second_clock){
        if entity @s[tag=*moonlit_monolith.fixed] run function
*block.moonlit_monolith.ten_second_clock.main
}
```

9. Traduzioni

Aggiungere coppie-chiave valore per traduzioni senza accedere al file .json

```
lang."item.haywire.moonlit_monolith" = "Moonlit Monolith" // traduzione di default a inglese americano (en_us), che è anche fallback per tutte le traduzioni assenti nelle altre lingue
lang["it_it"]."item.haywire.moonlit_monolith" = "Monolite Lunare" // esempio di traduzione in italiano
```

normalmente si ha un unico file json per ogni lingua.

10. Suoni

Stessa cosa per i suoni

```
sounds["block.moonlit_monolith.moonstone_vanishing"] = {
    "sounds":[
      {
        "name":"haywire:block/moonlit_monolith/moonstone_vanishing"
      }
    ],
    "subtitle":
"subtitles.haywire.block.moonlit_monolith.moonstone_vanishing"
}
```

Un esempio di sounds.json scritto a mano dove si può notare quanto difficile sia orientarsi. Anche qui c'è un problema simile a quello del punto 4. Dato che texture e suoni sono nei loro file png e ogg rispettivamente, forse la soluzione migliore è avere due cartelle apposta per texture e suoni.

11. Sintassi più veloce per macro

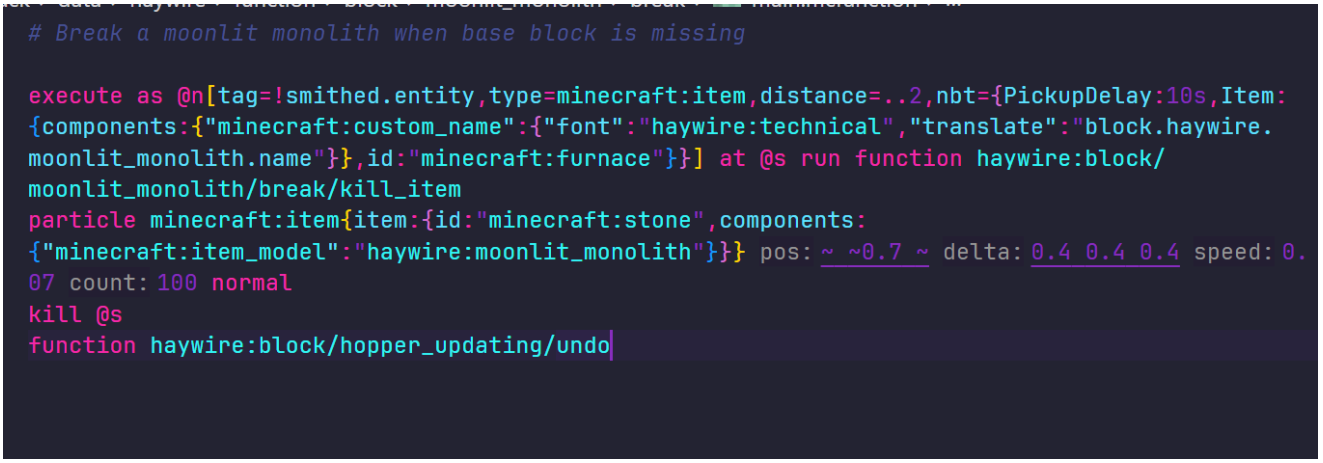
```
with storage *temp root.macro_input {
    $data remove storage *storage
root.balemoon[{moonlit_monolith:$(string_uuid)}]
}
// esempio con dati "costanti"
with {string_uuid:"0a602d40-a11f-43ec-808e-6db195d3913e"} {
    $data remove storage *storage
root.balemoon[{moonlit_monolith:$(string_uuid)}]
}
```

anziché

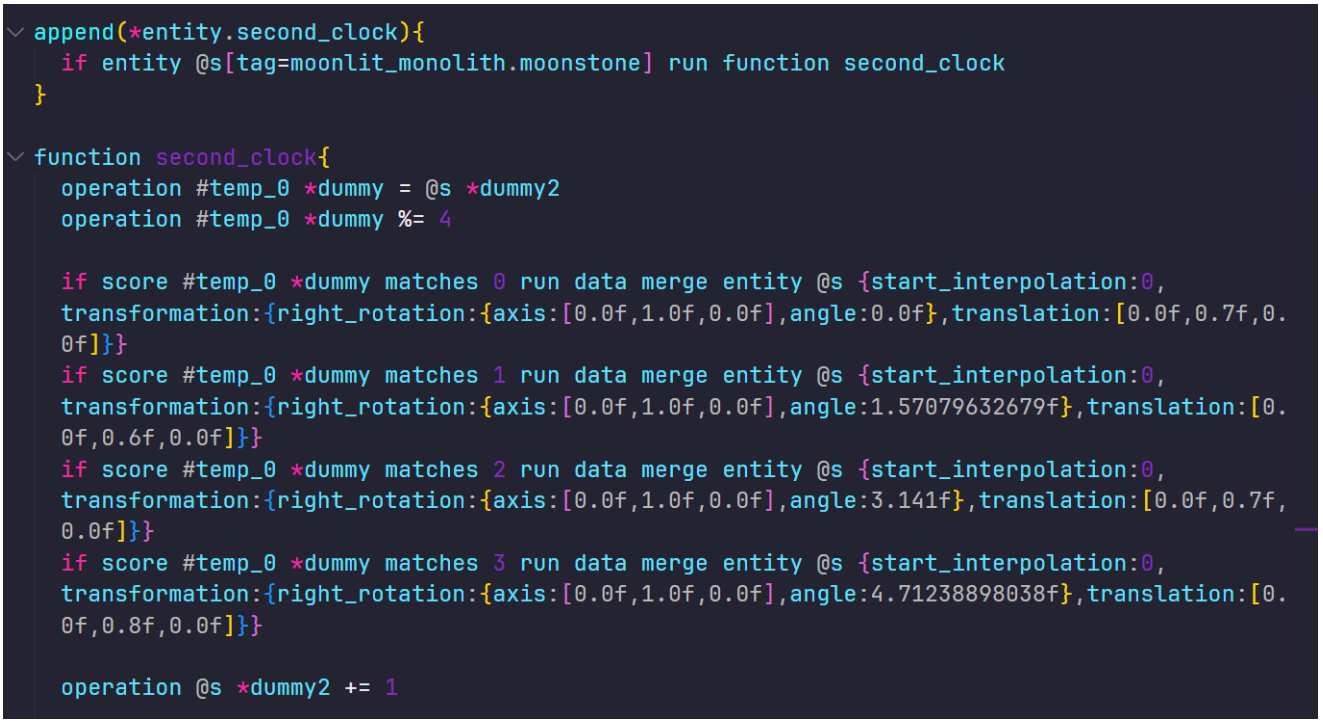
```
// remove storage.mcfunction
$data remove storage haywire:storage
root.balemoon[{moonlit_monolith:$(string_uuid)}]

// chiamata alla funzione con macro
function remove_storage with storage haywire:temp root.macro_input
```

12. Syntax highlighting (non importantissima)

mcfunction ha un estensione vscode per autocompletamento e syntax highlighting.



se invece imposto vscode per applicare il syntax highlighting di GO ai file con estensione .mcf (estensione dei file con questo linguaggio ideale) si ha qualcosa di affine all'immagine sopra.



13. Scorciatoie if/unless, if-elseif

- `if data block ~ ~ ~ {Items:[{components:{"minecraft:custom_data":{*{placed_block:1b,block:"moonlit_monolith"}}}}]} positioned ~ ~0.5 ~ run ...` sintassi classica ma con `execute` omesso

- `if (predicate in_the_expanse && block ~ ~-2 ~ vault) return run function place_fixed` : omissione execute e raggruppamento delle condizioni tra parentesi tonde. Inoltre uso `&&` al posto di un altro `if`
- `if(entity @s[tag=*moonlit_monolith.night] &! predicate night) function day` : dato che non si possono invertire gli esiti di controlli, ma solo controllare che non siano accaduti (con `unless`) al posto di scrivere `&& !<condizione>` si scrive `&!` (equivalente ad `unless`, che letteralmente significherebbe "e non")
- `if ... else` e `if ... else if ... else` : dovrebbe essere possibile implementarli usando `return` per fermare il flusso di esecuzione
  Mi sono reso conto ora che mi sono dimenticato di dire che return può anche restituire comandi (nel senso che esegue il comando specificato dopo return e poi interrompe il flusso di esecuzione della funzione) e non solo int come avevo detto all'ultimo incontro, cosa che torna utile in questo caso

```
say doing things
if(cond1) say 1
else if(cond2) say 2
else say 3
say do more things
```

diventa

```
say doing things
function if_else1
say do more things

// if_else1.mcfunction:
execute if cond1 run return run say 1
execute if cond2 run return run say 2
say 3
```

questo blocco if-else va messo nella sua funzione perché altrimenti i comandi dopo al return non verrebbero eseguiti

```
// cond2 = true
say doing things
execute if cond1 run return run say 1
execute if cond2 run return run say 2 // l'esecuzione termina qua
say 3
say do more things // questo che dovrebbe essere eseguito lostesso non viene eseguito per via del return chiamato prima
```

inoltre, se i comandi da eseguire se è verificata una condizione sono più di uno va creata una funzione per ciascuno di essi

```
say doing things
if(cond1){
    say 1
    say this is the if
}
else if(cond2){
    say 2
    say this is the else if
}
else{
    say 3
    say this is the default
}
say do more things
```

diventa

```
say doing things
function if_else1
say do more things

// if_else1.mcfunction:
execute if cond1 run return run function cond1
execute if cond2 run return run function cond2
// comandi chiamati solo se i primi due if sono falsi
say 3
say this is the default


// cond1.mcfunction
say 1
say this is the if

// cond2.mcfunction
say 2
say this is the if else
```

per lo stesso motivo di prima

```
execute if cond1 run return run say 1 // comando eseguito, poi interrompe il flusso di esecuzione della funzione
execute if cond1 run return run say this is the if // mai raggiungibile
```

14. Commenti multilinea

Mcfunction supporta commenti: ogni riga che inizia con `#` è considerata un commento, ed ingorata a compilazione (non si può dichiarare un commento dopo un comando nella stessa riga).
io preferirei uno stile di commento simile a c/java, che usa `//` per la singola riga, che possono essere anche messi sulla stessa linea del codice, e `/* ... */` per i commenti multilinea.

15. Prelievo texture da spritesheet e texture animate

Chi crea le texture che usa nei suoi progetti, spesso lo fa su un file png che ne contiene molte (in modo da avere più prototipi, provare stili e colori diversi), e poi copia una specifica texture nel suo file. Questo continuo copia incolla è noioso specialmente se si modifica una texture, che poi deve essere nuovamente copiata nel suo file singolo.
Per questo dato che sto gia "processando" le texture per copiarle da una cartella del progetto alla loro destinazione finale (`textures/...` -> `<resourcepack>/assets/<namespace>/textures/...`)
si può fare un ulteriore passaggio e al posto di scegliere una texture cosi
`texture item.stone_glyph_illager = textures/item/stone_glyph_illager`
si potrebbe scrivere
`texture item.stone_glyph_illager = textures/item/sprites(3,2,w,h)`
dove dall'immagine sprites, si seleziona quella alla terza riga e seconda colonna (ogni riga e colonna è composta da 16 pixel, che è la risoluzione di default delle texture e la dimensione di immagine minima per non rompere il mipmapping).
`w` e `h` sono due parametri opzionali usati per indicare la larghezza e l'altezza della selezione, sempre in multipli di 16 (0,0,1,5 = prima riga, prima colonna, largo 16 pixel verso destra e alto 80 verso il basso). Questo perché certe texture non sono grandi 16x16 e tantomeno con forma quadrata.
Le texture animate (non presenti nel progetto) sono costituite da più frame disposti verticalmente nello stesso png, quindi una da 5 frame sarà in un png lungo 16px e alto 80.
Le proprietà dell'animazione sono definite in un file strutturato come un json chiamato esattamente come l'immagine a cui si fa riferimento seguito da `.mcmeta` (`a.png` -> `a.png.mcmeta`).
La mia è di definire un metodo per le texture che permette loro di definire un animazione.
Simile al punto 8:

```
texture item.my_animated_texture = textures/item/my_animated_texture
animation(my_animated_texture) = { "animation": { "frames": [ { "index": 0,
"time": 20 }, 0, 1, 2, 3, 4 ] } }
```

16. Utilizzo e riferimento a variabili locali

Nel file `item/stone_glyph/illager.mcf` ho definito una strutta json chiamata components, che poi ho utilizzato sia nella loot table, che nella ricetta, dato che sia se creo l'oggetto o lo

trovo in un baule, voglio che i dati associati ad esso siano li stessi. Questo concetto è da approfondire meglio, questo è solo un esempio.

17. Implementazione feature di linguaggi di programmazione

17.1 Ciclo for

Nel estratto di codice che ho fatto non mi è capitato di doverlo utilizzare, ma ci sono casi in cui farebbe comodo generare comandi uguali dove cambia solo un numero. Ad esempio per modificare tutti gli oggetti nell'inventario di un giocatore bisogna modificare applicare le modifiche slot per slot (ce ne sono 36 in totale).
Quindi sarebbe comodo poter scrivere

```
function change_all_items{
    for(int i=0;i<37;i++){
        if(items entity @s container.{i} stone_sword){
        item replace entity @s container.{i} with iron_sword
        }
    }
}
```

per sostituire, dove presenti, le spade in pietra di un giocatore con quelle in ferro.

17.2 Variabili locali + ciclo for + lookup table

Come detto in precedenza si usano delle lookup table per generare alcuni valori di funzioni matematiche, che poi vengono estratti con macro.
Sarebbe comodo poterli generare senza doverli andare a cercare altrove semplicemente con

```
function make_sin{
    table = []
    for (int angle = 1; angle <= 360; angle++) {
            table[i] = Math.sin(angle)
        }
    data modify storage *lookup_tables sin set value {table}
}
```

Nel prossimo incontro vorrei approfondire meglio delle linee guida per una sintassi migliore che integra mcfunction con feature di linguaggi veri e propri.