Bozza del 22/10/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Framework per la Meta-programmazione di Minecraft

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

Bozza del 22/10/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Framework per la Meta-programmazione di Minecraft

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

Bozza del 22/10/2025

Sommario

In questo documento tratterò del mio lavoro svolto sotto la supervisione del prof. Padovani nello sviluppare un sistema software che agevola l'utilizzo della *Domain Specific Language* del videogioco *Minecraft*.
Inizialmente verranno illustrate la struttura e i principali componenti di questa DSL, evidenziandone gli aspetti sintattici e strutturali che ne determinano le principali criticità. Successivamente sarà presentato l'approccio adottato per mitigare tali problematiche, utilizzando una libreria Java sviluppata durante il tirocinio. Tale libreria è stata progettata con l'obiettivo di semplificare le operazioni più ripetitive e onerose, sfruttando i costrutti di un linguaggio ad alto livello e consentendo, e anche di definire più oggetti all'interno di un unico file, favorendo così uno sviluppo più coerente e strutturato.
Attraverso un *working example* verrà poi mostrato come tale libreria consenta di ridurre la complessità nello sviluppo dei punti più critici, mettendola a confronto con l'approccio tradizionale.
Infine, mostrerò la differenza in termini di righe di codice e file creati tra i due sistemi, con l'intento di affermare l'efficienza della mia libreria.

Bozza del 22/10/2025

Indice dei contenuti

Introduzione

Se non fosse per il videogioco *Minecraft*, non sarei qui ora. Quello che per me nel 2014 era un modo di esprimere la mia creatività costruendo con cubi in un mondo tridimensionale, si è rivelato presto essere il luogo dove per anni ho scritto ed eseguito i miei primi frammenti di codice.
Motivato dalla mia abilità nel saper programmare in questo linguaggio non banale, ho perseguito una carriera di studio in informatica.

Pubblicato nel 2012 dall'azienda svedese Mojang, *Minecraft* è un videogioco appartenente al genere *sandbox*, famoso per l'assenza di una trama predefinita, in cui è il giocatore stesso a costruire liberamente la propria esperienza e gli obiettivi da perseguire.
Come suggerisce il nome, le attività principali consistono nello scavare per ottenere risorse e utilizzarle per creare nuovi oggetti o strutture. Il tutto avviene all'interno di un ambiente tridimensionale virtualmente infinito.

Proprio a causa dell'assenza di regole predefinite, fin dal suo rilascio *Minecraft* era dotato di un insieme rudimentale di comandi che consentiva ai giocatori di aggirare le normali meccaniche di gioco, ad esempio ottenendo risorse istantaneamente o spostandosi liberamente nel mondo. Con il tempo, tale meccanismo è diventato un articolato linguaggio di configurazione e scripting, basato su file testuali, che costituisce una *Domain Specific Language* (DSL) attraverso la quale sviluppatori di terze parti possono modificare numerosi aspetti e comportamenti dell'ambiente di gioco.

Minecraft è sviluppato in Java, ma questa DSL, chiamata *mcfunction*, adotta un paradigma completamente diverso. Essa non consente di introdurre nuovi comportamenti intervenendo direttamente sul codice sorgente: le funzionalità aggiuntive vengono invece definite attraverso gruppi di comandi, interpretati dal motore interno di *Minecraft* (e non dal compilatore Java), ed eseguiti solo al verificarsi di determinate condizioni. In questo modo l'utente percepisce tali funzionalità come parte integrante dei contenuti originali del gioco. Negli ultimi anni, grazie all'introduzione e all'evoluzione di una serie di file in formato JSON, è progressivamente diventato possibile creare esperienze di gioco quasi completamente nuove. Tuttavia, il sistema presenta ancora diverse limitazioni, poiché gran parte della logica continua a essere definita e gestita attraverso i file *mcfunction*.

Il tirocinio ha avuto come obiettivo la progettazione e realizzazione di un sistema che semplifica la creazione, sviluppo e distribuzione di questi file, creando un ambiente di sviluppo unificato. Esso consiste in una libreria Java che permette di definire la gerarchia dei file in un sistema ad albero tramite oggetti. Una volta definite tutte le *feature*, esegue il programma per ottenere un progetto pronto per l'uso.

Il risultato è un ambiente di sviluppo più coerente e accessibile, che permette di integrare *feature* di Java in questa DSL, per facilitare la scrittura e gestione dei file.

Nel prossimo capitolo verrà presentata la struttura generale del sistema, descrivendone gli elementi principali e il loro funzionamento. In seguito verrà fatta un'analisi delle principali problematiche e limitazioni del sistema, insieme a una rassegna delle soluzioni proposte nello stato dell'arte. Successivamente sarà illustrata la struttura e implementazione della mia libreria, accompagnata da un *working example* volto a mostrare in modo concreto il funzionamento del progetto. L'ultimo capitolo sarà dedicato all'analisi dei risultati ottenuti e delle possibili evoluzioni future.

Struttura e Funzionalità di un Pack

2.1. Cos'è un Pack

I file JSON e *mcfunction* devono trovarsi in specifiche cartelle per poter essere riconosciuti dal compilatore di *Minecraft* ed essere integrati nel videogioco. La cartella radice che contiene questi file si chiama *datapack*.
Un *datapack* può essere visto come la cartella `java` di un progetto Java: contiene la parte che detta i comportamenti dell'applicazione.

Come i progetti Java hanno la cartella `resources`, anche *Minecraft* dispone di una cartella in cui inserire le risorse. Questa si chiama *resourcepack*, e contiene principalmente font, modelli 3D, *texture*, traduzioni e suoni.
Con l'eccezione di *texture* e suoni, i quali permettono l'estensione `png` e `ogg` rispettivamente, tutti gli altri file sono in formato JSON.
Le *resourcepack* sono state concepite e rilasciate prima dei *datapack*, con lo scopo di dare ai giocatori un modo di sovrascrivere le *texture* e altri *asset* del videogioco. Gli sviluppatori di

datapack hanno poi iniziato ad utilizzare *resourcepack* per definire le risorse che il progetto da loro sviluppato avrebbe richiesto.

L'insieme di *datapack* e *resourcepack* è chiamato *pack*. Questo, riprendendo il parallelismo precedente, corrisponde all'intero progetto Java, e sarà poi la cartella che verrà pubblicata o condivisa.

2.2. Struttura e Componenti di Datapack e Resourcepack

All'interno di un *pack*, *datapack* e *resourcepack* hanno una struttura molto simile.

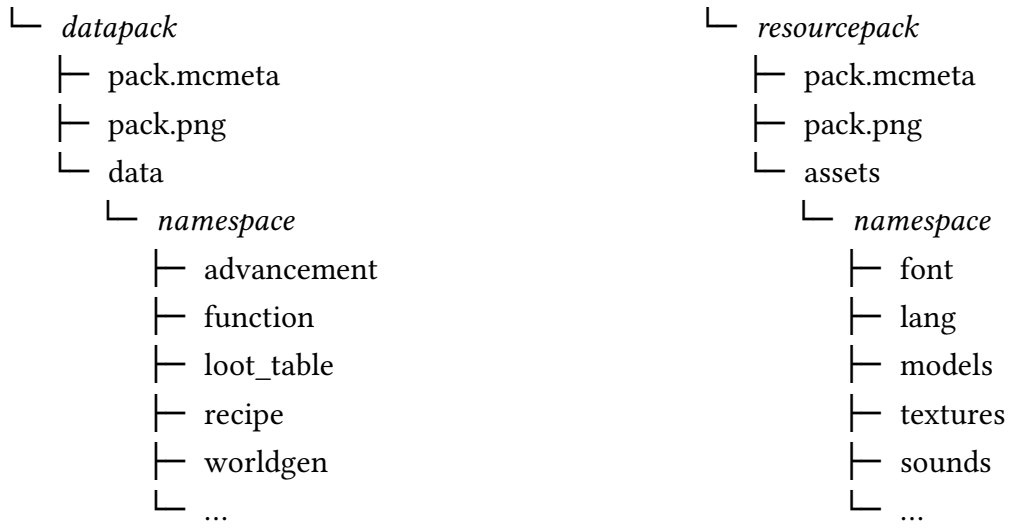

Figura 1: *datapack* e *resourcepack* a confronto.

Anche se l'estensione non lo indica, il file `pack.mcmeta` è in realtà scritto in formato JSON e definisce l'intervallo delle versioni (chiamate *format*) supportate dalla cartella, che con ogni aggiornamento di *Minecraft* variano, e non corrispondono all'effettiva *game version*.
Ad esempio, per la versione 1.21.10 del gioco, il `pack_format` dei *datapack* è 88 e quello delle *resourcepack* è 69. Queste possono cambiare anche settimanalmente, se si stanno venendo rilasciati degli *snapshot*[1].

Ancora più rilevanti sono le cartelle al di sotto di `data` e `assets`, chiamate *namespace*. Se i progetti Java seguono la seguente struttura `com.package.author`, allora i *namespace* possono essere visti come la sezione `package`.

[1]Con il termine snapshot si indicano le versioni di sviluppo intermedie del gioco, rilasciate periodicamente per testare le modifiche in arrivo nei futuri aggiornamenti.

> This isn't a new concept, but I thought I should reiterate what a «namespace» is. Most things in the game has a namespace, so that if we add `something` and a mod (or map, or whatever) adds `something`, they're both different `something`s. Whenever you're asked to name something, for example a loot table, you're expected to also provide what namespace that thing comes from. If you don't specify the namespace, we default to `minecraft`. This means that `something` and `minecraft:something` are the same thing.
>
> — Nathan Adams[2]

I *namespace* sono fondamentali per evitare che i file omonimi di un *pack* sovrascrivano quelli di un altro. Per questo, in genere i *namespace* o sono abbreviazioni o coincidono con il nome stesso progetto che si sta sviluppando, e si usa lo stesso per *datapack* e *resourcepack*.
Tuttavia, si vedrà come operare in *namespace* distinti è sia sufficiente a garantire l'assenza di conflitti tra i diversi *pack*, poiché questi vengono spesso installati dagli utenti in gruppo.

Il namespace `minecraft` è riservato alle risorse native del gioco: sovrascriverle comporta il rischio di rimuovere funzionalità originali o di alterare il comportamento previsto del gioco. È interessante notare che anche gli sviluppatori di *Minecraft* stessi fanno uso dei *datapack* per definire e organizzare molti comportamenti del gioco, come definire le risorse che si possono ottenere da un baule, o gli ingredienti necessari per creare un certo oggetto. In altre parole, i *datapack* non sono solo uno strumento a disposizione dei giocatori per personalizzare l'esperienza, ma costituiscono anche il **meccanismo interno attraverso cui il gioco stesso struttura e gestisce alcune delle sue funzionalità principali**.
Bisogna specificare che i domandi e file `.mcfunction` non sono utilizzati in alcun modo dagli sviluppatori della Mojang per implementare funzionalità del videogioco. Come precedentemente citato, tutta la logica è dettata da codice Java.

All'interno dei *namespace* si trovano directory i cui nomi identificano in maniera univoca la natura e la funzione dei contenuti al loro interno: se metto un file JSON che il compilatore riconosce come `loot_table` nella cartella `recipe`, il questo segnalerà un errore e il file non sarà disponibile nella sessione di gioco.

In `function` si trovano file e sottodirectory con testo in formato *mcfunction*. Questi si occupano di far comunicare tutte le parti di un *pack* tra loro tramite una serie di funzioni contenenti comandi.

[2] Sviluppatore di *Minecraft* parte del team che implementa *feature* inerenti a *datapack*.

2.3. Comandi

Prima di spiegare cosa fanno i comandi, bisogna definire gli elementi basi su cui essi agiscono. In *Minecraft*, si possono creare ed esplorare mondi generati in base a un *seed* casuale. Ogni mondo è composto da *chunk*, colonne dalla base di 16x16 cubi, e altezza di 320.
L'unità più piccola in questa griglia è il blocco, la cui forma coincide con quella di un cubo di lato unitario. Ogni blocco in un mondo è dotato di collisione ed individuabile tramite coordinate dello spazio tridimensionale. Si definiscono entità invece tutti gli oggetti dinamici che si spostano in un mondo: sono dotate di una posizione, rotazione e velocità.

I dati persistenti di blocchi ed entità sono memorizzati in una struttura dati ad albero chiamata *Named Binary Tags* (NBT). Il formato «stringificato», `SNBT` è accessibile agli utenti e si presenta come una struttura molto simile a JSON, formata da coppie di chiave e valori.

```snbt
{
  name1: 123,
  name2: "foo",
  name3: {
    subname1: 456,
    subname2: "bar"
    },
  name4: [
    "baz",
    456,
    {
      subname3: "bal"
    }
  ]
}
```

Codice 1: Esempio di `SNBT`.

Un comando è un'istruzione testuale che Minecraft interpreta per eseguire una specifica azione, come assegnare oggetti al giocatore, modificare l'ora del giorno o creare entità. Molti comandi usano selettori per individuare l'entità su cui essere applicati o eseguiti.

```mcfunction
say @e[
  type = player
]
```

Codice 2: Esempio di comando che tra tutte le entità, stampa quelle di tipo giocatore.

Sebbene non disponga delle funzionalità tipiche dei linguaggi di programmazione di alto livello come cicli `for` e `while`, strutture dati complesse o variabili generiche, il sistema

dei comandi fornisce comunque strumenti che consentono di riprodurre alcuni di questi comportamenti in forma limitata.

I comandi che più si avvicinano ai concetti tipici della programmazione sono:

2.3.1. Scoreboard

`scoreboard` permette di creare dizionari di tipo `<Entità, Objective>`. Un `objective` rappresenta un valore intero a cui è associata una condizione (*criteria*) che ne determina la variazione. Il *criteria* `dummy` corrisponde ad una condizione vuota, irrealizzabile. Su questi valori è possibile eseguire operazioni aritmetiche di base, come l'aggiunta o la rimozione di un valore costante, oppure la somma, sottrazione, moltiplicazione e divisione con altri `objective`. Dunque una *scoreboard* può essere meglio vista come un dizionario `<Entità,<Intero, Condizione>>`.
Prima di poter eseguire qualsiasi operazione su di essa, una *scoreboard* deve essere inizializzata. Questo viene fatto con il comando
`scoreboard objectives add <objective> <criteria>`.
Per eseguire operazioni che non dipendono da alcuna entità, si usano i cosiddetti *fakeplayer*. Al posto di usare nomi di giocatori o selettori, si prefiggono i nomi con caratteri illegali, quali `$` e `#`. In questo modo ci si assicura che un valore non sia associato ad un vero utente.

```mcfunction
scoreboard objectives add my_scoreboard dummy
scoreboard players set #20 my_scoreboard 20
scoreboard players set #val my_scoreboard 100
scoreboard players operation #val my_scoreboard /= #20 my_scoreboard
```

Codice 3: Esempio di operazioni su una *scoreboard*, equivalente a `int val = 100; val /= 20;`

Dunque, il sistema delle *scoreboard* permette di creare ed eseguire operazioni semplici esclusivamente su interi, con *scope* globale, se e solo se fanno parte di una *scoreboard*.

2.3.2. Data

`data` consente di ottenere, modificare e combinare i dati NBT associati a entità, blocchi e *storage*. Come menzionato in precedenza, il formato NBT, una volta compresso, viene utilizzato per la persistenza dei dati di gioco. Oltre alle informazioni relative a entità e blocchi, in questo formato vengono salvati anche gli *storage*. Questi sono un modo efficiente di immagazzinare dati arbitrari senza dover dipendere dall'esistenza di un certo blocco o entità. Per prevenire i conflitti, ogni *storage* dispone di una *resource location*, che convenzionalmente coincide con il *namespace*. Vengono dunque salvati come `command_storage_<namespace>.dat`.

```mcfunction
data modify storage my_namespace:storage name set value "My Cat"
data merge entity @n[type=cat] CustomName from storage my_namespace:storage name
data remove storage my_namespace:storage name
```

Codice 4: Esempio di operazioni su dati NBT

Questi comandi definiscono la stringa `My Cat` nello *storage*, successivamente combinano il valore dallo *storage* al campo nome dell'entità gatto più vicina, e infine cancellano i dati impostati.

2.3.3. Execute

`execute` consente di eseguire un altro comando cambiando valori quali l'entità esecutrice e la posizione. Questi elementi definiscono il contesto di esecuzione, ossia l'insieme dei parametri che determinano le modalità con cui il comando viene eseguito. Si usa il selettore `@s` per fare riferimento all'entità del contesto di esecuzione corrente.
Tramite `execute` è anche possibile specificare condizioni preliminari e salvare il risultato dell'esecuzione. Dispone inoltre di 14 sottocomandi, o istruzioni, che posso essere raggruppate in 4 categorie:

- modificatori: cambiano il contesto di esecuzione;
- condizionali: controllano se certe condizioni sono rispettate;
- contenitori: salvano i valori di output di un comando in una *scoreboard*, o in un contenitore di NBT;
- `run`: esegue un altro comando.

Tutti questi sottocomandi possono essere concatenati e usati più volte all'interno di uno stesso comando `execute`.

```mcfunction
execute as @e
  at @s
  store result score @s on_stone
  if block ~ ~-1 ~ stone
```

Codice 5: Esempio di comando `execute`.

Questo comando sta definendo una serie di passi da fare;

1. per ogni entità (`execute as @e`);
2. sposta l'esecuzione alla loro posizione attuale (`at @s`);
3. salva l'esito nello score `on_stone` di quell'entità;
4. del controllo che, nella posizione corrente del contesto di esecuzione, il blocco sottostante sia di tipo `stone`.

Al termine dell'esecuzione, la *scoreboard* `on_stone` di ogni entità sarà 1 se si trovava su un blocco di pietra, 0 altrimenti.

2.4. Funzioni

Le funzioni sono insiemi di comandi raggruppati all'interno di un file *mcfunction*, una funzione non può esistere se non in un file `.mcfunction`. A differenza di quanto il nome possa suggerire, non prevedono inerentemente valori di input o di output, ma contengono uno o più comandi che vengono eseguiti in ordine.
Le funzioni possono essere invocate in vari modi da altri file di un datapack:

- tramite comandi: `function namespace:function_name` esegue la funzione subito, mentre `schedule namespace:function_name <delay>` la esegue dopo un certo tempo specificato.
- da *function tag*: una *function tag* è una lista in formato JSON di riferimenti a funzioni. *Minecraft* ne fornisce due nelle quali inserire le funzioni da eseguire rispettivamente ogni game loop (`tick.json`)[3], e ogni volta che si ricarica da disco il datapack (`load.json`). Queste due *function tag* sono riconosciute dal compilatore di *Minecraft* solo se nel namespace `minecraft`.
- Altri oggetti di un *datapack* quali `Advancement` (obiettivi) e `Enchantment` (incantesimi).

Le funzioni vengono eseguite durante un game loop, completando tutti i comandi che contengono, inclusi quelli invocati altre funzioni. Le funzioni usano il contesto di esecuzione dell'entità che le sta invocando (se presente). Quando un comando `execute` altera il contesto di esecuzione, la modifica non influenza i comandi successivi, ma viene propagata alle funzioni chiamate a partire da quel punto.

In base alla complessità del branching e alle operazioni eseguite dalle funzioni, il compilatore (o più precisamente, il motore di esecuzione dei comandi) deve allocare una certa quantità di risorse per svolgere tutte le istruzioni durante un singolo tick. Il tempo di elaborazione aggiuntivo richiesto per l'esecuzione di un comando o di una funzione è definito *overhead*.

Le funzioni possono includere linee *macro*, ovvero comandi che preceduti dal simbolo `$`, hanno parte o l'intero corpo sostituito al momento dell'invocazione da un oggetto NBT indicato dal comando invocante.

`main.mcfunction` `mcfunction`

```
1  function foo:macro_test {value:"bar"}
2  function foo:macro_test {value:"123"}
```

`macro_test.mcfunction` `mcfunction`

```
1  $say my value is $(value)
```

Codice 6: Esempio di chiamata di funzione con *macro*.

[3] Il game loop di *Minecraft* viene eseguito 20 volte al secondo; di conseguenza, anche le funzioni incluse nel tag `tick.json` vengono eseguite con la stessa frequenza.

Il primo comando di `main.mcfunction` stamperà `my value is bar`, il secondo `my value is 123`.

L'esecuzione dei comandi di una funzione può essere interrotta dal comando `return`. Funzioni che non contengono questo comando possono essere considerate di tipo `void`. Tuttavia il comando return può solamente restituire la parola chiave `fail` o un intero predeterminato, a meno che non si usi una *macro*.

Una funzione può essere richiamata ricorsivamente, anche modificando il contesto in cui viene eseguita. Questo comporta il rischio di creare chiamate senza fine, qualora la funzione si invochi senza alcuna condizione di arresto. È quindi responsabilità del programmatore definire i vincoli alla chiamata ricorsiva.

iterate.mcfunction `mcfunction`

```
1 particle flame ~ ~ ~
2 execute if entity @p[distance=..10] positioned ^ ^ ^0.1 run function
  foo:iterate
```

Codice 7: Esempio di funzione ricorsiva che crea una scia lunga 10 blocchi nella direzione dove il giocatore sta guardando.

Questa funzione ogni volta che viene chiamata creerà una piccola *texture* intangibile e temporanea (*particle*), alla posizione in cui è invocata la funzione. Successivamente controlla se è presente un giocatore nel raggio di 10 blocchi. In caso positivo si sposta il contesto di esecuzione avanti di $\frac{1}{10}$ di blocco e si chiama nuovamente la funzione. Quando il sotto-comando `if` fallisce, la funzione non sarà più eseguita.

Un linguaggio di programmazione si definisce Turing completo se soddisfa tre condizioni fondamentali:

- Presenta rami condizionali: deve poter eseguire istruzioni diverse in base a una condizione logica. Nel caso di *mcfunction*, ciò è realizzabile tramite il sotto-comando `if`.
- È dotato di iterazione o ricorsione: deve consentire la ripetizione di operazioni. In questo linguaggio, tale comportamento è ottenuto attraverso la ricorsione delle funzioni.
- Permette la memorizzazione di dati: deve poter gestire una quantità arbitraria di informazioni. In *mcfunction*, ciò avviene tramite la manipolazione dei dati all'interno dei *storage*.

Pertanto, *mcfunction* può essere considerato a tutti gli effetti un linguaggio Turing completo. Tuttavia, come verrà illustrato nella sezione successiva, sia il linguaggio stesso sia il sistema di file su cui si basa presentano diverse limitazioni e inefficienze. In particolare, l'esecuzione di operazioni relativamente semplici richiede un numero considerevole di righe di codice e di file, che in un linguaggio di più alto livello potrebbero essere realizzate in modo molto più conciso.

Problemi pratici e limiti tecnici

Il linguaggio *mcfunction* non è stato originariamente concepito come un linguaggio di programmazione Turing completo. Nel 2012, prima dell'introduzione dei *datapack*, il comando `scoreboard` veniva utilizzato unicamente per monitorare statistiche dei giocatori, come il tempo di gioco o il numero di blocchi scavati. In seguito, osservando come questo e altri comandi venissero impiegati dalla comunità per creare nuove meccaniche e giochi rudimentali, gli sviluppatori di *Minecraft* iniziarono ampliare progressivamente il sistema, fino ad arrivare, nel 2017, alla nascita dei *datapack*.

Ancora oggi l'ecosistema dei *datapack* è in costante evoluzione, con *snapshot* che introducono periodicamente nuove funzionalità o ne modificano di già esistenti. Tuttavia, il sistema presenta ancora diverse limitazioni di natura tecnica, dovute al fatto che non era stato originariamente progettato per supportare logiche di programmazione complesse o essere utilizzato in progetti di grandi dimensioni.

3.1. Limitazioni di Scoreboard

Come è stato precedentemente citato, `scoreboard` è usato per eseguire operazioni su interi. Operare con questo comando tuttavia presenta numerosi problemi.

Innanzitutto, oltre a dover creare un *objective* prima di poter eseguire operazioni su di esso, è necessario assegnare le costanti che si utilizzeranno, qualora si volessero eseguire operazioni di moltiplicazione e divisione. Inoltre, un singolo comando `scoreboard` prevede una sola operazione.

Di seguito viene mostrato come l'espressione `int x = (y*2)/4-2` si calcola in *mcfunction*. Le variabili saranno prefissate da `$`, e le costanti da `#`.

```mcfunction
scoreboard objectives add math dummy
scoreboard players set $y math 10
scoreboard players set #2 math 2
scoreboard players set #4 math 4
scoreboard players operation $y math *= #2 math
scoreboard players operation $y math /= #4 math
scoreboard players remove $y math 2
scoreboard players operation $x math = $y math
```

Operazioni su `$y`

Codice 8: Esempio con $y = 10$

Qualora non fossero stati impostati i valori di `#2` e `#4`, il compilatore li avrebbe valutati con valore 0 e l'espressione non sarebbe stata corretta.

Si noti come, nell'esempio precedente, le operazioni vengano eseguite sulla variabile y, il cui valore viene poi assegnato a x. Di conseguenza, sia `#x` math che `#y` conterranno il risultato finale pari a 3. Questo implica che il valore di y viene modificato, a differenza dell'espressione a cui l'esempio si ispira, dove y dovrebbe rimanere invariato. Per evitare questo effetto collaterale, è necessario eseguire l'assegnazione $x = y$ prima delle altre operazioni aritmetiche.

```mcfunction
scoreboard objectives add math dummy
scoreboard players set $y math <some value>
scoreboard players set #2 math 2
scoreboard players set #4 math 4
scoreboard players operation $x math = $y math
scoreboard players operation $x math *= #2 math
scoreboard players operation $x math /= #4 math
scoreboard players remove $x math 2
```

Operazioni su `$x`

Codice 9: Esempio di espressione con `scoreboard`

La soluzione è quindi semplice, ma mette in evidenza come in questo contesto non sia possibile scrivere le istruzioni nello stesso ordine in cui verrebbero elaborate da un compilatore tradizionale.

Un ulteriore caso in cui l'ordine di esecuzione delle operazioni e il dominio ristretto agli interi assumono particolare rilevanza riguarda il rischio di errori di arrotondamento nelle operazioni che coinvolgono valori prossimi allo zero.

Si supponga si voglia calcolare il 5% di 40. Con un linguaggio di programmazione di alto livello si ottiene 2 calcolando `40/100*5` e `40*5/100`. Scomponendo queste operazioni in comandi `scoreboard` si ottiene rispettivamente:

```mcfunction
scoreboard players operation set $val math 40
scoreboard players operation $val math /= #100 math
scoreboard players operation $val math *= #5 math
```

```mcfunction
scoreboard players operation set $val math 40
scoreboard players operation $val math *= #5 math
scoreboard players operation $val math /= #100 math
```

Codice 10: Calcolo della percentuale con ordine di operazioni invertito

Nel primo caso, poiché $\frac{40}{100} = 0$ nel dominio degli interi, il risultato finale sarà 0: nella riga 3, infatti, viene eseguita l'operazione 0×5.
Nel secondo caso invece, si ottiene il risultato corretto pari a 2, poiché le operazioni vengono eseguite nell'ordine $40 \times 5 = 200$ e successivamente $\frac{200}{100} = 2$.

3.2. Assenza di Funzioni Matematiche

Poiché tramite *scoreboard* è possibile eseguire esclusivamente le quattro operazioni aritmetiche di base, il calcolo di funzioni più complesse quali logaritmi, esponenziali, radici quadrate o funzioni trigonometriche risulta particolarmente difficile da implementare.

Bisogna inoltre considerare il fatto che queste operazioni saranno ristrette al dominio dei numeri naturali.
Si può dunque cercare un algoritmo che approssimi queste funzioni, oppure creare una *lookup table*.

```mcfunction
scoreboard players set #sign math -400
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
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
execute if score #sign math matches 400 run scoreboard players add .out math 1
```

Codice 11: Algoritmo che approssima la funzione $\sin(x)$.

La scrittura di algoritmi di questo tipo è impegnativa, e spesso richiede di gestire un input moltiplicato per 10^n il cui output è un intero dove sia assume che le ultime n cifre siano decimali[4]. Inoltre, questo approccio può facilmente provocare problemi di *integer overflow*.

Dunque, in seguito all'introduzione delle *macro*, si sono iniziate ad utilizzare delle *lookup table*. Queste sono *array* salvati in *storage* che contengono tutti gli output di una certa funzione in un intervallo prefissato.

Ipotizziamo mi serva la radice quadrata con precisione decimale di tutti gli interi tra 0 e 100. Si può creare uno *storage* che contiene i valori $\sqrt{i}\ \forall i \in [0, 100] \cap \mathbb{N}$.

```mcfunction
data modify storage my_storage sqrt set value [
  0,
  1.0,
  1.4142135623730951,
  1.7320508075688772,
  2.0,
…                                   …
  10.0
]
```

Codice 12: *Lookup table* per $\sqrt{x}$, con $0 \leq x \leq 100$.

Dunque, data `get storage my_storage sqrt[4]` restituirà il quinto elemento dell'array, ovvero 2.0, l'equivalente di $\sqrt{4}$.

[4] Solitamente $n = 3$.

Dato che sono richiesti gli output di decine, se non centinaia di queste funzioni, i comandi per creare le *lookup table* vengono generati con script Python, ed eseguiti da *Minecraft* solamente quando si ricarica il *datapack*, dato che queste strutture non sono soggette ad operazioni di scrittura, solo di lettura.

3.3. Alto Rischio di Conflitti

Nella sezione precedente è stato modificato lo *storage* `my_storage` per inserirvi un array. Si noti che non è stato specificato alcun *namespace*, per cui il sistema ha assegnato implicitamente quello predefinito, `minecraft:`.

Qualora un mondo contenesse due *datapack* sviluppati da autori diversi, ed entrambi modificassero `my_storage` senza indicare esplicitamente un *namespace*, potrebbero verificarsi conflitti.

Un'altra situazione che può portare a conflitti è quando due *datapack* sovrascrivono la stessa risorsa nel *namespace* `minecraft`. Se entrambi modificano `minecraft/loot_table/blocks/stone.json`, che determina gli oggetti si possono ottenere da un blocco di pietra, il compilatore utilizzerà il file del *datapack* che è stato caricato per ultimo.

Il rischio di sovrascrivere o utilizzare in modo improprio risorse appartenenti ad altri *datapack* non riguarda solo gli elementi che prevedono un *namespace*, ma si estende anche a componenti come *scoreboard* e *tag*.

In questo esempio sono presenti due *datapack*, sviluppati da autori diversi, con lo stesso obiettivo: eseguire una funzione relativa all'entità chiamante (`@s`) al termine di un determinato intervallo di tempo. In entrambi i casi, le funzioni incaricate dell'aggiornamento del timer vengono eseguite ogni *tick*, ovvero venti volte al secondo.

`timer_a.mcfunction` `mcfunction`

```
1 scoreboard players add @s timer 1
2 execute if score @s timer matches 20 run function some_function
```

`timer_b.mcfunction` `mcfunction`

```
1 scoreboard players remove @s timer 1
2 execute if score @s timer matches 0 run function some_function
```

Codice 13: Due funzioni che aggiornano un timer.

Le due funzioni modificano lo stesso *fakeplayer* all'interno dello stesso *scoreboard*. Poiché `timer_a` incrementa `timer` e `timer_b` lo decrementa, al termine di un *tick* il valore rimane invariato. Se invece entrambe variassero `timer` nello stesso verso, ad esempio incrementandolo, la durata effettiva del timer risulterebbe dimezzata. Questo è uno dei motivi per cui il nome di una *scoreboard* deve essere prefissato con un *namespace*, ad esempio `a.timer`[5].

Tra le varie condizioni per cui i selettori possono filtrare entità, ci sono i *tag*, ovvero stringhe memorizzate in un array nell'NBT di un entità.

Di conseguenza, se nell'esempio precedente gli sviluppatori intendono che la funzione `timer` venga eseguita esclusivamente dalle entità contrassegnate da un determinato *tag*, ad esempio `has_timer`, i comandi per invocare `timer_a` e `timer_b` risulteranno i seguenti:

tick_a.mcfunction — mcfunction

```
1  execute as @e[tag=has_timer] run function a:timer_a
```

tick_b.mcfunction — mcfunction

```
1  execute as @e[tag=has_timer] run function b:timer_b
```

In entrambi i casi, `@e[tag=has_timer]` seleziona lo stesso insieme di entità. Ciò può risultare problematico se, allo scadere del timer di *b*, vengono eseguiti comandi che determinano comportamenti inaspettati o erronei per le entità del *datapack* di *a* (o viceversa).

Dunque, come per i nomi delle *scoreboard*, è buona norma prefissare i tag con il *namespace* del proprio progetto.

In conclusione, è buona pratica utilizzare prefissi anche per i nomi di *storage*, *scoreboard* e *tag*, nonostante i *datapack* compilano correttamente anche senza di essi.

3.4. Assenza di Code Blocks

Nei linguaggi come C o Java, i blocchi di codice che devono essere eseguiti condizionalmente o all'interno di un ciclo vengono racchiusi tra parentesi graffe. In Python, invece, la stessa funzione è ottenuta tramite l'indentazione del codice.

In una funzione *mcfunction*, questo non si può fare. Se si vuole eseguire una serie di comandi condizionalmente, è necessario creare un altro file che li contenga, oppure ripetere la stessa

[5] Come separatore si usa `.` e non `:` in quanto quest'ultimo è un carattere supportato nel nome di una *scoreboard*.

condizione su più righe. Quest'ultima opzione comporta maggiore *overhead*, specialmente quando il comando viene eseguito in più *tick*.

Di seguito viene riportato un esempio di come si può scrivere un blocco `if-else`, o `switch`, sfruttando il comando `return` per interrompere il flusso di esecuzione del codice nella funzione corrente.

```mcfunction
execute if entity @s[type=cow] run return run say I'm a cow
execute if entity @s[type=cat] run return run say I'm a cat
say I'm neither a cow or a cat
```

Codice 15: Funzione che in base all'entità esecutrice, stampa un messaggio diverso.

In questa funzione, i comandi dalla riga 2 in poi non verranno mai eseguiti se il tipo dell'entità è cow. Se la condizione alla riga 1 risulta falsa, l'esecuzione invece procede alla riga successiva, dove viene effettuato un nuovo controllo sul tipo dell'entità; anche in questo caso, se la condizione è soddisfatta, l'esecuzione si interrompe.

```
switch(entity){
  case "cow" -> print("I'm a cow")
  case "cat" -> print("I'm a cat")
  default -> print("I'm neither a cow or a cat")
}
```

Codice 16: Pseudocodice equivalente alla funzione precedente.

La funzione è abbastanza intuitiva, e corrisponde a qualcosa che si vedrebbe in un linguaggio di programmazione di alto livello. Ipotizziamo ora che si vogliano eseguire due o più comandi in base all'entità.

```mcfunction
execute if entity @s[type=cow] run return run say I'm a cow
execute if entity @s[type=cow] run return run say moo

execute if entity @s[type=cat] run return run say I'm a cat
execute if entity @s[type=cat] run return run say meow

say I'm neither a cow or a cat
```

Codice 17: Funzione errata per eseguire più comandi data una certa condizione.

Ora, se l'entità è di tipo `cow`, il comando alla riga 2 non verrà mai eseguito, anche se la condizione sarebbe soddisfatta. Dunque, è necessario creare una funzione che contenga quei due comandi.

`main.mcfunction`

```mcfunction
execute if entity @s[type=cow] run return run function is_cow
```

```
2 execute if entity @s[type=cat] run return run function is_cat
3
4 say I'm neither a cow or a cat
```

`is_cow.mcfunction` `mcfunction`

```
1 say I'm a cow
2 say moo
```

`is_cat.mcfunction` `mcfunction`

```
1 say I'm a cat
2 say meow
```

Considerando che i *datapack* si basano sull'esecuzione di funzioni **in base a eventi già esistenti**, sono numerosi i casi in cui ci si trova a creare più file che contengono un numero ridotto, purché significativo, di comandi.

Per quanto riguarda i cicli, come mostrato in Codice 7, l'unico modo per ripetere gli stessi comandi più volte è attraverso la ricorsione. Di conseguenza, ogni volta che è necessario implementare un ciclo, è indispensabile creare almeno una funzione dedicata. Se è invece necessario un contatore per tenere traccia dell'iterazione corrente (il classico `i` dei cicli `for`), è possibile utilizzare funzioni ricorsive che si richiamano passando come parametro una *macro*, il cui valore viene aggiornato all'interno del corpo della funzione. In alternativa, si possono scrivere esplicitamente i comandi necessari a gestire ciascun valore possibile, in modo analogo a quanto avviene con le *lookup table*.

Ipotizziamo si voglia determinare in quale *slot* dell'inventario del giocatore si trovi l'oggetto `diamond`. Una possibile soluzione è utilizzare una funzione che iteri da 0 a 35 (un giocatore può tenere fino a 36 oggetti diversi), dove il parametro della *macro* indica lo *slot* che si vuole controllare, ma questo approccio comporta un overhead maggiore rispetto alla verifica diretta, caso per caso, dei valori da 0 a 35.

`find_diamond.mcfunction` `mcfunction`

```
1  execute if items entity @s container.0 diamond run return run say slot 0
2  execute if items entity @s container.1 diamond run return run say slot 1
…                                        …
36 execute if items entity @s container.35 diamond run return run say slot
   35
```

In questa funzione, la ricerca viene interrotta da `return` appena si trova un diamante, ed è stato provato che abbia un *overhead* minore della ricorsione. Come nel caso delle *lookup table*, i file che fanno controlli di questo genere vengono creati script Python.

Infine, Codice 6 dimostra che, per utilizzare una *macro*, è sempre necessario creare una funzione capace di ricevere i parametri di un'altra funzione e applicarli a uno o più comandi indicati con `$`. Questa è probabilmente una delle ragioni più valide per cui scrivere una nuova funzione; tuttavia, va comunque considerata nel conteggio complessivo dei file la cui creazione non è necessaria in un linguaggio di programmazione ad alto livello.

Dunque, programmando in *mcfunction* è necessario creare una funzione, ovvero un file, ogni volta che si necessiti di:
- un blocco `if-else` che esegua più comandi;
- un ciclo;
- utilizzare una *macro*.

Ciò comporta un numero di file sproporzionato rispetto alle effettive righe di codice. Tuttavia, ci sono altre problematiche relative alla struttura delle cartelle e dei file nello sviluppo di *datapack* e *resourcepack*.

3.5. Organizzazione e Complessità della Struttura dei File

I problemi mostrati fin'ora sono prettamente legati alla sintassi dei comandi e ai limiti delle funzioni, tuttavia non sono da trascurare il quantitativo di file di un progetto.

Affinché *datapack* e *resourcepack* vengano riconosciuti dal compilatore, essi devono trovarsi rispettivamente nelle directory `.minecraft/saves/<world_name>/datapacks` e `.minecraft/resourcepacks`. Tuttavia, operare su queste due cartelle in modo separato può risultare oneroso, considerando l'elevato grado di interdipendenza tra i due sistemi. Lavorare direttamente dalla directory radice `.minecraft/` invece inoltre poco pratico, poiché essa contiene un numero considerevole di file e cartelle non pertinenti allo sviluppo del *pack*.

Una possibile soluzione consiste nel creare una directory che contenga sia il *datapack* sia il *resourcepack* e, successivamente, utilizzare *symlink* o *junction* per creare riferimenti dalle rispettive cartelle verso i percorsi in cui il compilatore si aspetta di trovarli.
I *symlink* (collegamenti simbolici) e le *junction* sono riferimenti a file o directory che consentono di accedere a un percorso diverso come se fosse locale, evitando la duplicazione dei contenuti.

Disporre di un'unica cartella radice contenente *datapack* e *resourcepack* semplifica notevolmente la gestione del progetto. In particolare, consente di creare una sola repository Git, facilitando così il versionamento del codice, il tracciamento delle modifiche e la collaborazione tra più sviluppatori.

Attraverso il sistema delle *release* di GitHub è possibile ottenere un link diretto a *datapack* e *resourcepack* pubblicati, che può poi essere utilizzato nei principali siti di hosting. Queste piattaforme, essendo spesso gestite da piccoli team di sviluppo, tendono ad affidarsi a servizi esterni per la memorizzazione dei file, come GitHub o altri provider.

Ipotizzando di operare in un ambiente di lavoro unificato, come quello illustrato in precedenza, viene presentato un esempio di struttura che mostra i file necessari per introdurre un nuovo *item* (oggetto). Sebbene l'*item* costituisca una delle funzionalità più semplici da implementare, la sua integrazione richiede comunque un numero non trascurabile di file.

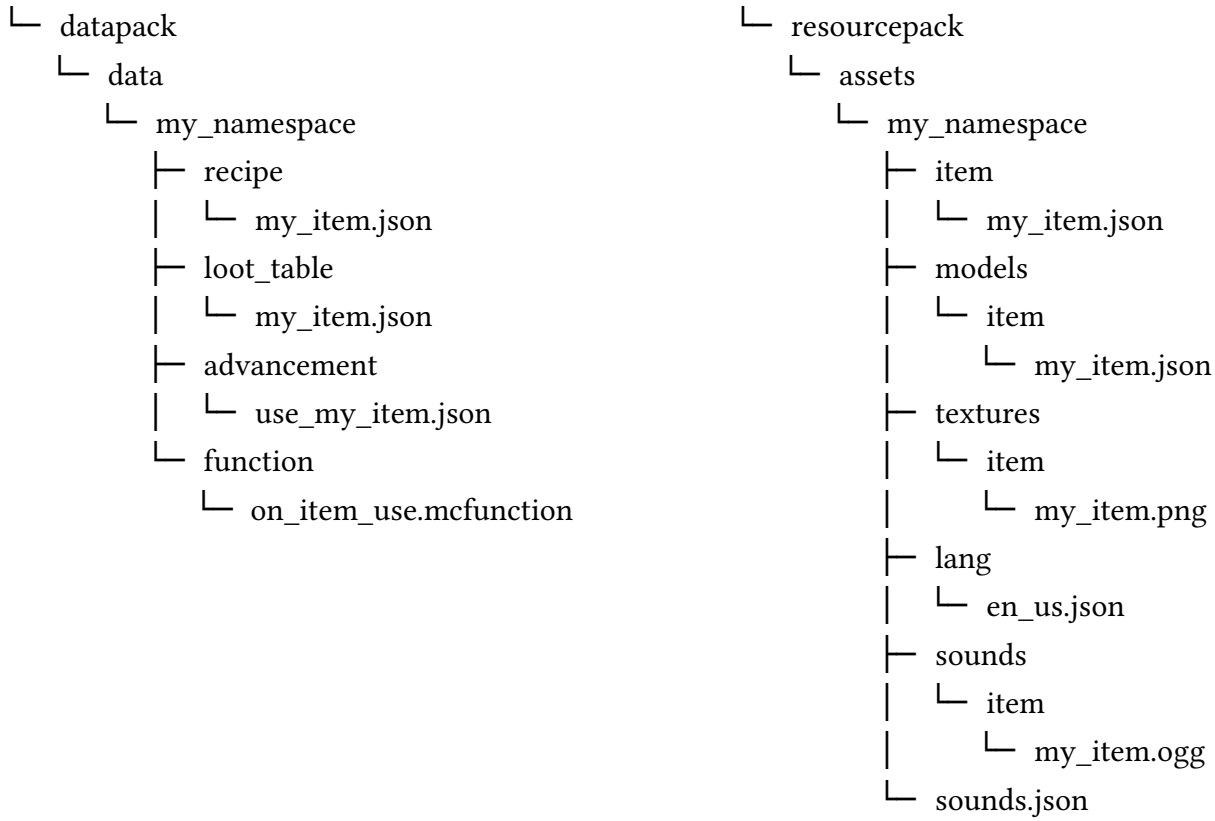

Figura 2: File necessari per implementare un semplice *item*.

Nella sezione *data*, che determina la logica e i contenuti, *loot_table* e *recipe* definiscono rispettivamente le proprietà dell'oggetto, e come questo può essere creato. L'*advancement* `use_my_item` serve a rilevare quando un giocatore usa l'oggetto, e chiama la funzione `on_item_use` che produrrà un suono.

I suoni devono essere collocati all'interno degli *assets*. Per poter essere riprodotti, ciascun suono deve avere un file audio in formato `.ogg` ed essere registrato nel file `sounds.json`. Nella cartella *lang* sono invece presenti i file responsabili della gestione delle traduzioni, organizzate come insiemi di coppie chiave-valore.
Per definire l'aspetto visivo dell'oggetto, si parte dalla sua *item model definition*, situata nella cartella `item`. Questa specifica il modello che l'*item* utilizzerà. Il modello 3D, collocato in `models/item`, ne definisce la forma geometrica, mentre la *texture* associata al modello è contenuta nella directory `textures/item`.

Si osserva quindi che, per implementare anche la *feature* più semplice, è necessario creare sette file e modificarne due. Pur riconoscendo che ciascun file svolge una funzione distinta e che la loro presenza è giustificata, risulterebbe certamente più comodo poter definire questo tipo di risorse *inline*.

Con il termine *inline* si intende la definizione e utilizzo una o più risorse direttamente all'interno dello stesso file in cui vengono impiegate. Questa modalità risulterebbe particolarmente vantaggiosa quando un file gestisce contenuti specifici e indipendenti. Ad esempio, nell'aggiunta di un nuovo item, il relativo modello e la *texture* non verrebbero mai condivisi con altri oggetti, rendendo superfluo separarli in file distinti.

Infine, l'elevato numero di file rende l'ambiente di lavoro complesso da navigare. In progetti di grossa portata questo implica, nel lungo periodo, una significativa quantità di tempo dedicata alla ricerca dei singoli file.

3.6. Stato dell'Arte delle Ottimizzazioni del Sistema

Alla luce delle numerose limitazioni di questo sistema, sono state rapidamente sviluppate soluzioni volte a rendere il processo di sviluppo più efficiente e accessibile.

In primo luogo, gli stessi sviluppatori di *Minecraft* dispongono di strumenti interni che automatizzano la generazione dei file JSON necessari al corretto funzionamento di determinate *feature*. Durante lo sviluppo, tali file vengono creati automaticamente tramite codice Java eseguito in parallelo alla scrittura del codice sorgente, evitando così la necessità di definirli manualmente.

Un esempio lampante è il file `sounds.json`, che registra i suoni definisce quali file `.ogg` utilizzare. Questo contiene quasi 25.000 righe di codice, creato tramite software che viene eseguito ogni volta che viene inserita una nuova *feature* che richiede un nuovo suono.

Tuttavia, questo software non è disponibile al pubblico, e anche se lo fosse, semplificherebbe la creazione solo dei file JSON, non *mcfunction*. Dunque, sviluppatori indipendenti hanno realizzato dei propri precompilatori, progettati per generare automaticamente *datapack* e *resourcepack* a partire da linguaggi o formati più intuitivi.

Un precompilatore è uno strumento che consente di scrivere le risorse e la logica di gioco in un linguaggio o formato più semplice, astratto o strutturato, e di tradurle automaticamente nei numerosi file JSON, *mcfunction* e cartelle richieste dal gioco.
Il precompilatore al momento più completo e potente si chiama *beet*, e si basa sulla sintassi di Python, integrata con comandi di *Minecraft*.
Questo precompilatore, come molti altri, presenta due criticità principali:

- Elevata barriera d'ingresso: solo gli sviluppatori con una buona padronanza di Python sono in grado di sfruttarne appieno le potenzialità;
- Assenza di documentazione: la mancanza di una guida ufficiale rende il suo utilizzo accessibile quasi esclusivamente a chi è in grado di interpretare direttamente il codice sorgente di *beet*.

Altri precompilatori forniscono un'interfaccia più intuitiva e un utilizzo più immediato al costo di completezza delle funzionalità, limitandosi a supportare solo una parte delle componenti che costituiscono l'ecosistema dei *pack*. Spesso, inoltre, la sintassi di questi linguaggi risulta più verbosa rispetto a quella dei comandi originali, poiché essi offrono esclusivamente un approccio programmatico alla composizione dei comandi senza portare ad alcun incremento nella loro velocità di scrittura.

```java
Execute myExecuteCommand = new Execute()
  .as("@a")
  .at("@s")
  .if("entity @s[tag=my_entity]")
  .run("say hello")
```

Questo è più articolato rispetto alla sintassi tradizionale `execute as @a at @s if entity @s[tag=my_entity] run say hello`.

La mia Implementazione

4.1. Approccio al Problema

Dato il contesto descritto e le limitazioni degli strumenti esistenti, ho cercato una soluzione che permettesse di ridurre la complessità d'uso senza sacrificare la completezza delle funzionalità. Di seguito verranno illustrate le principali decisioni progettuali e le ragioni che hanno portato alla scelta del linguaggio di sviluppo.

Inizialmente, su suggerimento del prof. Padovani, ho tentato di progettare un superset di *mcfunction*, ossia un linguaggio che estende quello originale introducendo nuove funzionalità mantenendone però la compatibilità. Questo linguaggio avrebbe consentito di dichiarare e utilizzare più elementi (*mcfunction* e JSON), all'interno di un unico file, arricchendo anche la sintassi con elementi di zucchero sintattico volti a semplificare la scrittura delle parti più verbose.

```mcf
package foo
scoreboard players operation @s var *= 4
if score @s var matches 10.. run function {
  say hello
  say something else
}
```

Codice 20: Esempio di questo *superset*, caratterizzato da file con l'estensione `.mcf`

Eseguendo questo codice, non solo si sarebbe creata la funzione dichiarata all'interno delle parentesi graffe, ma inserito il namespace prima di `var`, e creato il comando che assegna alla costante `#4` i valore 4. Come è stato mostrato nel Codice 8, per eseguire divisioni e moltiplicazioni per valori costanti, è prima necessario definirli in uno *score*. Compilando il frammento di codice dell'esempio, si sarebbero ottenuti i seguenti file:

```mcfunction
load.mcfunction
scoreboard players set #4 foo.var 4
```

```mcfunction
main.mcfunction
scoreboard players operation @s foo.var *= #4 foo.var
execute if score @s foo.var matches 10.. run function foo:5a3c50
```

```mcfunction
5a3c50.mcfunction
say hello
say something else
```

Ho inizialmente scelto di utilizzare la versione Java della libreria ANTLR per definire la grammatica del linguaggio. Tuttavia, mi sono presto reso conto che realizzare una grammatica in grado di cogliere tutte le sfumature della sintassi di *mcfunction*, integrandovi al contempo le mie estensioni, avrebbe richiesto un impegno di sviluppo superiore a quello compatibile con un progetto di tirocinio.

Ho quindi pensato di sviluppare una libreria che consentisse di definire la struttura di un *pack*, dalla radice del progetto fino ai singoli file, sotto forma di oggetti. In questo modo sarebbe stato possibile rappresentare l'intero insieme delle risorse come una struttura dati ad albero n-ario. Questa, al momento dell'esecuzione, sarebbe stata attraversata per generare automaticamente i file e le cartelle corrispondenti ai nodi, all'interno delle directory di *datapack* e *resourcepack*.

Il principale vantaggio di questo approccio consiste nella possibilità di definire più nodi all'interno dello stesso file, evitando così la frammentazione del codice e semplificando la gestione della struttura complessiva del *pack*. Inoltre, l'impiego di un linguaggio ad alto livello consente di sfruttare costrutti quali cicli e funzioni per automatizzare la generazione di comandi ripetitivi (ad esempio le già citate lookup table). Infine, la rappresentazione

a oggetti della struttura consente di definire metodi di utilità per accedere e modificare i nodi da qualsiasi punto del progetto. Ad esempio, si può implementare un metodo `addTranslation(key, value)` che permette di aggiungere, indipendentemente dal contesto in cui viene invocato, una nuova voce nel file delle traduzioni.

Dunque ho pensato a quale linguaggio di programmazione si potesse usare per realizzare questa libreria. Le mie opzioni erano Python e Java, e dopo aver valutato i loro punti di forza e debolezza, ho deciso di usare Java.

	Vantaggi	Svantaggi
Python	• Gestione semplice di stringhe (`f-strings`) e file JSON; • Sintassi concisa; • Facilmente distribuibile.	• Non nativamente orientato agli oggetti; • Tipizzazione dinamica che può causare errori a runtime; • Prestazioni inferiori in fase di esecuzione.
Java	• Maggiore familiarità con progetti di grandi dimensioni; • Completamente orientato agli oggetti; • Compilazione ed esecuzione più efficienti.	• Assenza di `f-strings` e manipolazione delle stringhe più complessa; • Gestione dei file JSON più verbosa; • Sintassi più prolissa, che rallenta la scrittura del codice.

Tabella 1: Java e Python a confronto.

Dopo un'attenta analisi, ho scelto di utilizzare Java per lo sviluppo del progetto, poiché secondo me è il mezzo ideale per l'applicazione di *design pattern* in grado di semplificare e rendere più robusta la fase di sviluppo, anche a costo di sacrificare parzialmente la comodità d'uso per l'utente finale.
Inoltre, il tipaggio statico di Java permette di identificare in fase di sviluppo eventuali utilizzi impropri di oggetti o metodi della libreria, consentendo anche agli utenti meno esperti di comprendere più facilmente il funzionamento del sistema.

4.2. Spiegazione basso livello

4.3. Spiegazione alto livello

4.4. Uso working example

Conclusione

Bibliografia