Bozza del 14/10/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Lorem ipsum dolor sit amet.

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

Bozza del 14/10/2025

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DIPARTIMENTO DI SCIENZA E INGEGNERIA

Corso di Laurea in Informatica per il Management

Lorem ipsum dolor sit amet.

Relatore:
Prof. Luca Padovani

Presentata da:
Alessandro Nanni

Sessione di Dicembre
Anno accademico 2024/2025

Bozza del 14/10/2025

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.
Verranno inizialmente illustrati i problemi sintattici e strutturali di questo ampio ecosistema di file.
Successivamente mostrerò come ho provato ad ovviarli, o almeno ridurli, tramite una libreria che si occupa di svolgere le operazioni più tediose e ripetitive. Tramite un *working example* esporrò in che modo ho semplificato lo sviluppo di punti critici, facendo confronti con l'approccio abituale.
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 14/10/2025

Indice dei contenuti

1
Introduzione

Se non fosse per il videogioco *Minecraft*, non sarei qui ora. Quello che per me inizialmente era un modo di esprimere la mia creatività piazzando 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.

Il sistema che inizialmente era stato pensato dagli sviluppatori della piattaforma come un modo di «barare» tramite comandi per ottenere oggetti istantaneamente e senza il minimo sforzo, si è col tempo evoluto in un ecosistema di file e codice che permette agli sviluppatori che decidono di usare questa *Domain Specific Language* per modificare moltissimi comportamenti dell'ambiente videoludico.

Minecraft è scritto in Java, ma questa DSL chiamata *mcfunction* è un linguaggio completamente diverso. Non fornisce agli sviluppatori il modo di aggiungere comportamenti nuovi, modificando il codice sorgente. Permette piuttosto di aggiungere *feature* aggiungendo frammenti di codice che vengono eseguiti solo sotto certe condizioni, dando ad un utilizzatore l'illusione che queste facciano parte dei contenuti classici del videogioco. Negli ultimi anni, in seguito ad aggiornamenti, tramite una serie di file JSON sta gradualmente diventando possibile creare esperienze del tutto nuove. Tuttavia questo sistema è ancora limitato, e gran parte della logica è comunque dettata dai file *mcfunction*.

1.1. 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 prima dei *datapack*, e permettevano ai giocatori sovrascrivere le *texture* e altri asset del videogioco. Gli sviluppatori di *datapack* hanno poi iniziato ad utilizzarle per definire nuove risorse, inerenti al progetto che stanno sviluppando.

Datapack e *resourcepack* formano il *pack* che, riprendendo il parallelismo precedente, corrisponde all'intero progetto Java. Questa sarà poi la cartella che verrà pubblicata.

1.2. Struttura di *datapack* e *resourcepack*

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

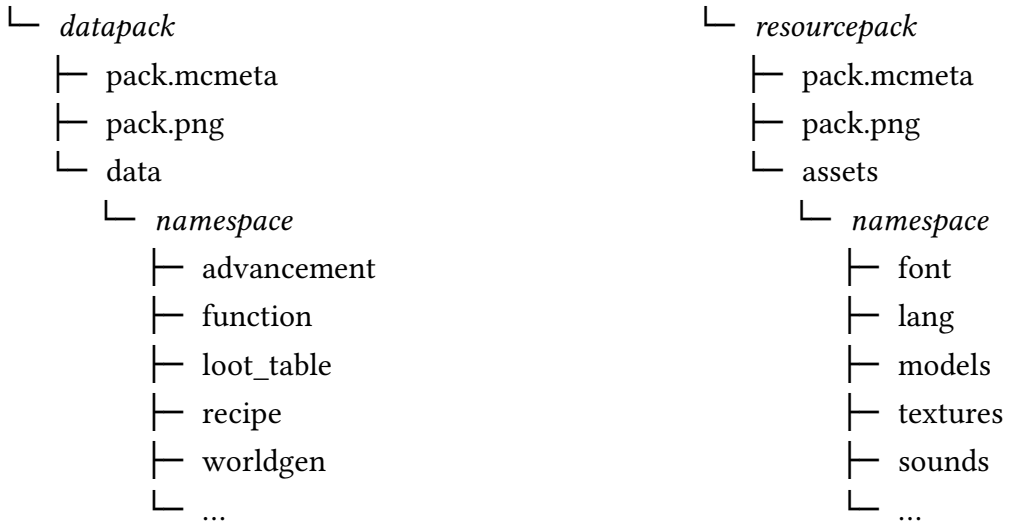

Figura 1: *datapack* e *resourcepack* a confronto.

Anche se l'estensione non lo indica, il file è in realtà scritto in formato JSON e definisce l'intervallo delle versioni (chiamate *format*) supportate dalla cartella, che con ogni aggiorna-

mento 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`.
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, in seguito si mostrerà come operare in namespace diversi non è sufficiente l'assenza di conflitti tra i *pack*, che spesso vengono utilizzati in gruppo.

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 comandi.

1.3. 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.

[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.

```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:

1.3.1. 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`.
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*.

1.3.2. Data

`data` consente di ottenere, modificare e combinare i 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 Chicken"
data merge entity @n[type=chicken] 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 Chicken` nello *storage*, successivamente combinano il valore dallo *storage* al campo nome della gallina più vicina, e infine cancellano i dati impostati.

1.3.3. 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.
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, il valore `on_stone` di ogni entità sarà 1 se si trovava su un blocco di pietra, 0 altrimenti.

1.4. Funzioni

Le funzioni sono insiemi di comandi raggruppati all'interno di un file *mcfunction*. A differenza di quanto il nome possa suggerire, non prevedono parametri di input o di output, ma contengono 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 funzioni. *Minecraft* ne fornisce due nelle quali inserire le funzioni da eseguire ogni game loop (`tick.json`)[2], 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` (condizioni).

[2]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.

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 sta invocando la funzione. un comando `execute` può cambiare il contesto, ma non si applicherà a tutti i comandi a seguirlo.

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 termine 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*.

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 `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.

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 sottocomando `if` fallisce, la funzione non sarà più eseguita.

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

- 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`.
- Iterazione o ricorsione: deve consentire la ripetizione di operazioni. In questo linguaggio, tale comportamento è ottenuto attraverso la ricorsione delle funzioni.
- 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.

1.5. Problemi e Limitazioni

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.

1.5.1. Limiti 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 `#`.

```
1 scoreboard objectives add math dummy                    mcfunction
2 scoreboard players set $y math 10
3 scoreboard players set #2 math 2
4 scoreboard players set #4 math 4
5 scoreboard players operation $y math *= #2 math
6 scoreboard players operation $y math /= #4 math
7 scoreboard players remove $y math 2
8 scoreboard players operation $x math = $y math
```

(1) 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.

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

(1) 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.

2
Come agevolare lo sviluppo

3
La mia implementazione

4
Conclusione

4
Bibliografia