Università di Modena e Reggio
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Triennale

Proposta per la prototipizzazione digitale di generici giochi da tavolo

Laureando:
Alessandro Mezzogori

Relatore:
Maurizio Vincini

A.A. 2021/2022

Università di Modena e Reggio
Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Triennale

Proposta per la prototipizzazione digitale di generici giochi da tavolo

Laureando:
Alessandro Mezzogori

Relatore:
Maurizio Vincini

A.A. 2021/2022

Sommario

Introduzione

Negli ultimi anni, si è presentata una notevole evoluzione dei meccanismi principali sfruttati dai creatori dei giochi da tavolo nell'ideazione di esperienze di gioco immersive, originali e stimolanti.
I nuovi generi sono caratterizzati da delle specifiche meccaniche e game flow (l'intero flusso di gioco): gli *engine building* come "Terraforming Mars" e "Gizmos", che spingono a sviluppare le interazioni tra i vari componenti acquisiti durante la partita per creare un motore con un solo obiettivo; gli strategici "Scythe" o il classico "Scacchi", che si concentrano sull'abilità dei giocatori di pianificare e adattarsi ai cambiamenti dell'ambiente di gioco; i cooperativi "Pandemic Legacy" e "Gloomhaven", che uniscono i giocatori per adempiere a un singolo compito comune; i piazziamento lavoratori come "Feast for Odin" e "Dune Imperium", che obbligano a scegliere con accortezza le poche mosse e l'utilizzo delle scarse risorse a disposizione.
Gli avanzamenti nello sviluppo dei giochi da tavolo generano con sé delle difficoltà nel descrivere logicamente e in maniera strutturata i flussi di gioco, limitando la possibilità di effettuare la prototipizzazione direttamente in modo digitale delle meccaniche di possibili nuovi giochi da tavolo, prima ancora di creare una versione fisica.
Nelle seguenti pagine si illustrerà il principale linguaggio impiegato attualmente, seguito da una nuova alternativa per la descrizione dei giochi da tavolo.
L'obiettivo finale è sia di fornire un'alternativa ai metodi in uso a oggi, sia di stimolare e ispirare la discussione sull'argomento del general game playing.
Importante notare che la proposta non si prospetta come definizione rigida dei requisiti dell'implementazione del linguaggio, ma come linea guida comune di partenza per futuri sviluppi e modifiche nella creazione di strumenti facilmente utilizzabili e apprendibili velocemente.

Game Description Language

Game Description Language é un formalismo di alto livello per descrivere le meccaniche e le regole di un generico gioco, basato sulla visione di quest'ultimo come una macchina a stati finiti.
È stato proposto da Michael Genesereth [2], professore nel dipartimento d'informatica all'università di Stanford, come parte del *General Game Playing Project*.
GGPP si pone l'obiettivo di sviluppare degli agenti generici che possano interagire efficacemente con un gioco sconosciuto tramite la sola descrizione in *Game Description Language* senza nessuno intervento umano.
In opposizione si trovano i modelli specializzati in un piccolo insieme di giochi come DeepMind *AlphaZero* [1] che conquistò scacchi, shogi e Go battendo alcuni tra migliori giocatori al mondo in tutti e tre.
La flessibilitá dei giocatori generali apre la possibilitá di creare dei modelli d'intelligenza artificiale che possano dedurre un metodo efficacie per risolvere qualsiasi tipologia di gioco, sia complessa come scacchi e go, ma anche semplice come tris.
In questo contesto *GDL* abilita gli agenti GGP (*General Game Playing Agents*) di avere un' unica sintassi capace di descrivere un qualsiasi ambiente di gioco, consentendo ai suddetti agenti di estrapolare le relazioni tra i vari nodi di sintassi del linguaggio, imparando i concetti di alto livello necessari per comprendere gli stati del gioco.

2.1 Sintassi

Esistono quattro classi di simboli: le costanti di oggetto, le costanti di funzione, le costanti di relazione e le variabili:

- Termine: é o una costante di oggetto o una variabile o un termine funzionale;
- Atomo: é un'espressione formata dalla relazione tra una costante e n termini;
- Letterale: un atomo o la negazione di un atomo;
- Regola: un'espressione composta da una testa, l'operatore `:-` e una congiunzione, marcata dall'operatore `&`, di zero o piú letterali.

role(r)	definisce un ruolo `r` nel gioco
base(p)	definisce una proposizione base `p` del gioco
input(r, a)	definisce `a` come un'azione del ruolo `r`
init(p)	definisce la proposizione `p` come vera nello stato iniziale del gioco
true(p)	definisce la proposizione `p` come vera nello stato corrente del gioco
does(r, a)	definisce che il giocatore `r` esegue l'azione `a`
next(p)	definisce la proposizione `p` come vera nello stato successivo
legal(r, a)	definisce che é legale per il ruolo `r` eseguire l'azione `a`
goal(r, n)	definisce che lo stato corrente ha utilità `n` per il giocatore `r`
terminale	definisce lo stato corrente come uno stato terminale

Inoltre per avere un programma legale si hanno le seguenti regole:

- Si richiede una definizione precisa di role, base, action e init;
- Bisogna definire legal, goal e terminal, in funzione di una relazione true;
- Bisogna definire next in termini di relazioni true e does;
- Bisogna che non ci siano regole con delle relazioni true o does nella testa.

Listing 2.1: Esempio: definizione cella di tris

```
base(cell(M,N,x)) :- index(M) & index(N)
base(cell(M,N,o)) :- index(M) & index(N)
base(cell(M,N,b)) :- index(M) & index(N)
```

Oltre alle primitive di gdl é possibile definire delle proprie relazioni come nell'esempio seguente:

Listing 2.2: colonnet e righe in tris

```
line(X) :- row(M,X)
line(X) :- column(M,X)

row(M,X) :-
  true(cell(M,1,X)) &
  true(cell(M,2,X)) &
  true(cell(M,3,X))

column(M,X) :-
  true(cell(1,M,X)) &
  true(cell(2,M,X)) &
  true(cell(3,M,X))
```

2.1.1 Estensioni

La sintassi originale di Game Description Language venne poi estesa da Michael Thielscher per le descrizioni di giochi a informazione incompleta [5] e per aggiungere l'introspezione al linguaggio [6]

2.2 Distanza dal giocatore umano

La semplicitá e forza nell'ambito del General Game Playing della Game Description Language e affini, uniti alla rapida espansione del machine learning ha creato un monopolio sui linguaggi di descrizione logica dei giochi, creando una carenza di uno specializzato nel descrivere i giochi dal lato della prototipizzazione, ricerca e utilizzo del prototipo tramite giocatore umano.
Le principali carenze di GDL come linguaggio di protipizzazione sono:

- é difficoltoso creare un gioco tramite la sua rappresentazione come macchina a stati finiti;
- un gioco in stato embrionale non avrá i concetti base ben delimitati, segue che non sará possibile definire le primitive del gioco in maniera efficacie;
- la mancanza di strutture e funzionalità base che sono condivise con la maggior parte dei giochi: carte, tabelloni, mazzi, ect...

Nel prossimo capitolo si esplorerá una proposta per un linguaggio di prototipizzazione, che tenta di risolvere le problematiche identificate:

- Tramite una rappresentazione del gioco tramite eventi e risposte a eventi;
- Senza bisogno di concetti ben definiti da utilizzare come fondamenta;
- Fornendo alcune implementazioni base di funzionalità e strutture dati.

Tabletop Game Prototype Language

Tabletop Game Prototype Language é una nuova possibile sintassi per un linguaggio specializzato nella prototipizzazione dei giochi da tavolo, proposto in questa tesi e illustrato nelle pagine successive.

3.1 Nozioni Fondamentali

3.1.1 Identificatori

Gli identificatori sono una qualsiasi stringa di caratteri che inizia per un trattino basso "_" o per lettera, opzionalmente seguiti da un qualsiasi ammontare di lettere o numeri senza spazi.
Un identificatore per essere legale deve essere unico all'interno del suo ambito di visibilità, questo significa che deve essere diverso sia dagli altri identificatori ma anche da operatori e clausole.

3.1.2 Operatori e Clausole

Gli operatori e le clausole sono delle parole chiave o simboli riservati che permettono rispettivamente la costruzione delle espressioni e delle istruzioni.

3.1.3 Blocchi

I blocchi sono un insieme struttura d'istruzioni, tag e altri blocchi. Un blocco si definisce come tutto il contenuto all'interno di una coppia di parentesi graffe, rispettivamente aperta e chiusa.
Il blocco delimita l'ambito, o scope, e la vita di una variabile definita al suo interno ovvero il suo campo d'azione, che parte dalla riga dov'è stata dichiarata fino alla fine del blocco che contiene la sua dichiarazione.
La visibilità di una variabile si estende a tutti gli ambiti figli del blocco di dichiarazione, dove figlio significa definito all'interno del padre (annidato).

3.1.4 Tags

I tag sono delle parole chiave riservate che modificano il comportamento del blocco associato, ogni tag descrive cosa è permesso definire al suo interno.
All'interno della proposta è possibile che si trovi la dicitura blocco tag come abbreviazione per indicare la coppia tag e il relativo blocco associato.

3.1.5 Commenti

I commenti sono linee di codice che vengono ignorate durante l'esecuzione del programma.
Per commentare una riga è sufficiente inserire una doppia barra "//", qualsiasi scritta oltre le due barre è considerata parte del commento.

3.2 Primitive del linguaggio

Tabletop Game Prototype Language si basa su un insieme di primitive da cui sono costruite tutte le funzionalità del linguaggio.

3.2.1 Oggetti

Un oggetto è un'istanza di una certa classe, un blocco di memoria che viene allocato e configurato secondo le specifiche definite dalla classe.
Gli oggetti hanno due comportamenti principali, ispirati dal linguaggio C# [4]:

Oggetti valore Detti anche value objects, si riferiscono direttamente al valore. Quando avviene una copia dell'oggetto, viene creata una nuova istanza della stessa tipologia di oggetto che contiene lo stesso valore. Il comportamento degli oggetti valore è utilizzato principalmente per i tipi primitivi come number, string, bool.

Oggetti riferimento Detti anche Reference objects, si riferiscono al blocco di memoria che gli è stato allocato, segue che, quando un reference object è copiato in una nuova variabile, nel nuovo oggetto si copierá il riferimento alla zona di memoria a cui l'oggetto originale si riferisce, evitando di allocare una nuova zona di memoria.
Una conseguenza importante del meccanismo dei riferimenti è che qualsiasi cambiamento effettuato su un oggetto riferimento sarà rispecchiato da tutti gli altri reference objects che possiedono lo stesso riferimento al blocco di memoria.

3.2.2 Tipi primitivi

I tipi primitivi sono degli oggetti valore particolari che rappresentano i valori puri del linguaggio come i numeri, le stringhe di testo, i valori booleani.

Numeri

I numeri sono rappresentati dal tipo primitivo "number" che accoglie un qualsiasi numero tale che sia compreso tra $\pm 5.0 * 10^{-324}$ e $\mp 1.7 * 10^{308}$ con una precisione tra le 15 e le 17 cifre, il tipo number equivale al tipo double presente in linguaggi come C, C#, ecc...

Stringa

Il tipo stringa, o string, rappresenta una qualsiasi successione di caratteri codificata in UTF-16, non ha una lunghezza massima predefinita. I valori string sono immutabili (ovvero non modificabili), all'esecuzione di espressioni che andrebbero a modificare il valore di una stringa, viene istanziata una nuova stringa con il valore modificato.

Booleano

Il tipo booleano, o bool, rappresenta una decisione binaria vera o falsa. Le variabili di tipo booleano dunque possono assumere due singoli valori rispettivamente rappresentati dalle parole chiave ”true” per il valore vero e ”false” per il valore falso.

Null

Il valore null non è un tipo primitivo in sé ma serve per indicare l'assenza del valore in una variabile. Ogni tipo può assumere il valore null.

3.2.3 Collezioni

Una collezione é un qualsiasi gruppo di oggetti che sono rappresentati come una singola unità, in altri linguaggi esistono varie tipologie di collezioni come array, hashtables, vectors, ect...
Attualmente sono supportate tre tipologie: lista, table, stack, la cui sintassi è ispirata dalle collezioni generiche del linguaggio C#

Liste

Le liste forniscono un insieme di oggetti accessibili tramite indici numerici interi, che mantiene l'ordine d'inserimento. [4]
Non hanno una dimensione fissa ma si adattano in base al numero di oggetti che contengono.
Per dichiarare una lista si utilizza il tipo lista, composto dalla parola chiave `List` e una coppia di parentesi angolate con al loro interno il tipo degli oggetti che si vuole raggruppare `List<<tipo>>`.
L'inizializzazione é effettuata tramite un blocco in cui è definibile un elenco di espressioni, dello stesso tipo definito nella dichiarazione, separate da una virgola.

```
List<<tipo>> <identificatore> = {
  <espressione tipo>,
  <espressione tipo>,
  ...
};
```

La lista accetta anche valori nulli, l'indice corrispondente presenterà un valore nullo quando si farà accesso. Le seguenti funzionalità sono supportate dalle liste:

Lunghezza	`<lista>.lenght`	calcola il numero d'indici occupati
Accesso	`<lista>[<numero>]`	accede all'indice numero, se non é intero la parte decimale verrà troncata.
Aggiungere una lista	`<lista>.append(<lista>)`	aggiunge tutti gli elementi di una lista in code
Rimuovere tutto	`<lista>.clear()`	rimuove tutti gli oggetti dalla lista
Contiene valore	`<lista>.contains(<oggetto>)`	ritorna true se la lista contiene lo stesso oggetto, falso altrimenti
Trova indice	`<lista>.indexOf(<oggetto>)`	restituisce l'indice del primo oggetto corrispondente, altrimenti -1
Rimuovi elemento	`<lista>.remove(<oggetto>)`	rimuove il primo oggetto corrispondente che trova, sposta il resto della lista avanti di una posizione
Rimuovi elemento all'indice	`<lista>.removeAt(<indice>)`	rimuove l'oggetto all'indice corrispondente, sposta il resto della lista avanti di una posizione
Ordinamento casuale	`<lista>.shuffle()`	mescola casualmente la lista
Copia	`<lista>.copy()`	crea una copia shallow della lista

Table

Ispirata dalla omonima struttura dati del famoso linguaggio di scripting LUA [3], sono array associativi, ovvero che possono essere indicizzati non solo con dei numeri ma con qualsiasi oggetto eccetto il valore null.
Per dichiarare una table si utilizza il tipo table, composto dalla parola chiave `Table` e una coppia di parentesi angolate con all'loro interno il tipo degli oggeti che si vuole raggruppare `Table<<tipo>>`.
L'inizializzazione é effettuata tramite un blocco in cui sono definibili delle coppie di espressioni separate da una virgola.

```
Table<<tipo>> <identificatore> = {
  {<espressione chiave>, <espressione valore>},
  {<espressione chiave>, <espressione valore>},
  ...
};
```

Le espressioni saranno valutate nel momento in cui la variabile sarà inizializzata, per avere una definizione legale della table le espressioni valore devono avere lo stesso tipo definito nella dichiarazione.
In caso una espressione chiave venga valutata a null, il programma terminerà con errore.
Le seguenti funzionionalità sono supportate per le table:

Funzione	Sintassi	Descrizione
Lunghezza	`<table>.length`	calcola quante coppie sono presenti all'interno
Accesso	`<table>[<chiave>]`	ritorna il valore o null se la chiave non é presente
Rimuovere tutto	`<table>.clear()`	elimina tutte le coppie contenute nella table
Copia	`<table>.copy()`	crea una copia shallow della table
Rimozione coppia	`<table>[<chiave>]` = null	rimuove una coppia chiave/valore dalla table

Stack

Gli stack sono una raccolta di oggetti di tipo LIFO (Last-in, First-out) [4]. Per dichiarare uno stack si utilizza il tipo stack, composto dalla parola chiave `Stack` e una coppia di parentesi angolate con all'loro interno il tipo degli oggetti che si vuole raggruppare `Stack<<tipo>>`.
L'inizializzazione é effettuata tramite un blocco in cui è definibile un elenco di espressioni, dello stesso tipo definito nella dichiarazione, separate da una virgola.

```
Stack<<tipo>> <identificatore> = {
  <espressione tipo>,
  <espressione tipo>,
  ...
};
```

Lo stack accetta valori nulli nelle funzioni di push ma non li aggiunge allo stack, trattandolo quindi come un noop (no operation).
Le seguenti funzionalità sono supportate:

Lunghezza	`<stack>.lenght()`	Restituisce il numero di oggetti nello stack
Inserimento	`<stack>.push(<tipo>)`	Inserisce in cima allo stack un oggetto
Inserimento in fronte	`<stack>.push_front(<tipo>)`	Inserisce in fronte allo stack un oggetto
Pop	`<stack>.pop()`	Rimuovare e restituisce l'oggetto in cima allo stack
Pop in fronte	`<stack>.pop_front()`	Rimuove e resistuisce l'oggeto in fronte allo stack
Copia	`<stack>.clone()`	Restituisce una copia shallow dello stack
Ordinamento casuale	`<stack>.shuffle()`	Mescola casualmente lo stack

3.2.4 Espressioni

Una espressione è una qualsiasi operazione che coinvolge zero o più operatori e uno o più operandi (o espressioni) che può essere valutata all'interno del linguaggio.
Ogni espressione ha un tipo di ritorno che indica il tipo dell'oggetto che sarà restituito dopo la valutazione dell'espressione.
Gli operatori sono predefiniti e si possono trovare tutti nella lista di definizione degli operatori.
Le espressioni vengono classificate in base al tipo di ritorno o al numero di operatori.
Nella definizione delle espressioni, con il tipo del valore si intende una qualsiasi espressione che ritorna un oggetto dello stesso tipo.

Espressioni letterali

Le espressioni letterali sono un qualsiasi valore puro di un tipo primitivo (number, string, bool).
Seguono alcuni esempi di espressioni letterali separate dalla virgola: 1, 0.2, "hello world", true, false

Espressioni matematiche

Le espressioni matematiche sono una qualsiasi operazione che coinvolga degli operatori matematici, hanno come tipo di ritorno il tipo number.
Le seguenti sono tutte espressioni matematiche:

- Addizione `<numero> + <numero>`
- Sottrazione `<numero> - <numero>`
- Prodotto `<numero> * <numero>`
- Divisione `<numero> / <numero>`
- Potenza `<numero> ^ <numero>`
- Modulo `<numero> % <numero>`

L'ordine di precedenza per la valutazione segue quello matematico:

1. Potenza
2. Prodotto, divisione e Modulo
3. Addizione e Sottrazione

Espressioni di confronto

Le espressioni di confronto coinvolgono gli operatori di maggioranza e minoranza, stretta e larga, il tipo di ritorno dell'espressione è un booleano il cui valore è il risultato del confronto:

- Minore di `<espressione> < <espressione>`
- Maggiore di `<espressione> > <espressione>`

- Minore uguale di `<espressione> <= <espressione>`
- Maggiore uguale di `<espressione> >= <espressione>`

Il tipo di ritorno dei due operandi deve essere il medesimo

Espressioni di uguaglianza

Le espressioni di uguaglianza coinvolgono gli operatori di uguaglianza, il tipo di ritorno dell'espressione è booleano il cui valore è il risultato dell'uguaglianza:

- Uguale a `<espressione> == <espressione>`
- Diverso da `<espressione> != <espressione>`

Come per le espressioni di confronto, il tipo dei due operandi deve essere il medesimo per avere una espressione di uguaglianza legale.

Espressioni logiche condizionali

Le espressioni logiche condizionali coinvolgono gli operatori logici AND e OR, il tipo di ritorno dell'espressione è booleano il cui valore è il risultato dell'operazione logica:

1. AND `<booleano> && <booleano>`
2. OR `<booleano> || <booleano>`

Le espressioni logiche condizionali cercano di valutare il minimo possibile per ottimizzare i controlli. Se una espressione si rivela vera o falsa alla valutazione del primo operando, l'operazione non valuterà il secondo operando ma ritornerà direttamente il corrispettivo risultato.

Espressioni logiche binarie

Le espressioni logiche binarie sono simili alle espressioni logiche condizionali ma valutano sempre entrambi gli operandi

1. AND `<booleano> & <booleano>`
2. OR `<booleano> | <booleano>`
3. XOR `<booleano> ^ <booleano>`
4. NOT `!<booleano>`

Espressioni di accesso

Le espressioni di accesso permettono d'interagire con i valori contenuti all'interno delle variabili. In base all'espressione si accederà a tipologie di oggetti diversi.

- Accesso a membro: `<oggetto>.<membro>`
  utilizza operatore di accesso `.` per accedere al valore di uno specifico membro di un oggetto. Il tipo di ritorno dell'espressione è il tipo del membro a cui si sta accedendo.

- Accesso tramite indice: `<collezione>[<chiave>]`
  utilizza l'operatore di accesso tramite un indice `[<chiave>]` per accedere alla variabile associata della lista.
  Ha come tipo di ritorno lo stesso tipo degli oggetti della lista.

- Accesso a variabile: `<identificatore>`
  l'identificatore unico deve appartenere ad variabile per poter essere legale. Accede al valore contenuto nella variabile, il tipo di ritorno dell'espressione è lo stesso tipo della variabile a cui si sta accedendo.

Espressione tipo

Ispirate dal linguaggio C# per la loro chiarezza, le espressioni tipo o type expression sono tutte quelle espressioni che si occupano della gestione dei tipi degli oggetti:

- Controllo del tipo: `<oggetto> is <tipo>`
  controlla se l'oggetto è del tipo richiesto. L'espressione ha come tipo di ritorno un valore booleano, il cui valore corrisponde il risultato del controllo: vero se l'oggetto è del tipo richiesto, falso altrimenti.

- Conversione del tipo, type casting: `<oggetto> as <tipo>`
  il type casting é utilizzato per convertire un oggetto in un oggetto di tipo diverso, il tipo di ritorno é lo stesso del tipo richiesto. Nel caso la conversione sia ammissibile l'espressione ristornerà il nuovo oggetto con il tipo prestabilito, in caso contrario sarà ritornato un valore nullo.

Espressioni d'istanziazione

L'espressione d'istanziazione `new <tipo>()` è usata per creare una nuova istanza del tipo richiesto, il tipo di ritorno corrisponde al tipo richiesto.
L'istanziazione di un oggetto può essere seguita da una lista d'inizializzazione per popolare velocemente con dei valori il nuovo oggetto.

Espressione d'assegnamento

`<espressione di accesso> = <espressione>`
L'espressione di assegnamento è utilizzata per assegnare il risultato di un'espressione a una variabile, attributo o a uno specifico indice di una lista.
Per avere un'espressione di assegnamento legale è necessario che i tipi di ritorno delle due espressioni siano gli stessi.
Il tipo di ritorno dell'espressione è nullo.

Espressione di chiamata

`<identificatore>(<lista argomenti>)`
L'espressione di chiamata è utilizzata per chiamare l'esecuzione della funzione con lo stesso identificatore con degli specifici argomenti definiti nella lista degli argomenti.
Il tipo di ritorno dell'espressione corrisponde al tipo di ritorno della funzione.
L'espressione di chiamata è approfondita in 3.5

Espressione giocatore

Una espressione giocatore, o player expression, è un'espressione speciale utilizzata per ritornare un oggetto giocatore.
Gli oggetti giocatori sono identificati tramite un numero partendo da zero fino al numero di giocatori meno uno, questi identificatori numerici possono essere utilizzati all'interno delle espressioni giocatore, che equivale a una espressione matematica il cui risultato è ristretto per poter essere mappato a un giocatore.
La funzione che si occupa di riportare il risultato dell'espressione matematica nell'intervallo di mappatura dei giocatori è la seguente

$$\begin{cases} r\%P & r >= 0 \\ (P - |r|\%P)\%P & r < 0 \end{cases}$$

Dove:

- r: è il risultato dell'equivalente espressione matematica
- P: è il numero di giocatori
- %: è l'operatore modulo

Ordine di precedenza

L'ordine standard (senza modificatori di precedenza) di valutazione delle espressioni è il seguente:

1. Espressioni d'istanziazione
2. Espressioni di chiamata
3. Espressioni di accesso
4. Espressioni matematiche
5. Espressioni giocatore
6. Espressioni di confronto
7. Espressioni di uguaglianza
8. Espressioni logiche binarie
9. Espressioni logiche condizionali
10. Espressioni di assegnamento

Se non specificato altrimenti le espressioni all'interno di una stessa categoria si risolvono da sinistra verso destra.
Qualsiasi espressione può essere inserita all'interno di parentesi tonde per aumentare la priorità di valutazione, la valutazione partirà delle espressioni maggiormente innestate.

3.2.5 Istruzioni

Le istruzioni, o statements, sono i macro comandi del linguaggio, ovvero una combinazione di clausole ed espressioni usate per adempire a un compito preciso. Le istruzioni devono terminare con il carattere `;`, ma nel caso sia possibile definire un blocco d'istruzioni associato è possibile omettere il `;`

Istruzione di ritorno

`return [<espressione>];`
L'istruzione di ritorno è utilizzata per finire l'esecuzione di un effetto, ritornando il controllo al codice che ha chiamato la funzione, restituendo (o no) un risultato in base al tipo di ritorno della funzione.
Il risultato restituito proviene dalla valutazione della espressione definita nella istruzione, nel caso si abbia un tipo di ritorno senza tipo bisogna omettere la espressione.

Istruzione espressione

`<espressione>;`
L'istruzione espressione, o expression statement, serve per eseguire una certa espressione;

Istruzione di dichiarazione

`<tipo> <identificatore> = <espressione>;`
L'istruzione di dichiarazione è utilizzata per definire delle nuove variabili del tipo richiesto e inizializzate con il risultato dell'espressione alla destra dell'operatore d'assegnamento.
Le variabili dichiarate sono utilizzabili dopo la loro dichiarazione tramite l'identificatore, ovvero tramite una espressione di accesso.

Istruzione di selezione

`if(<espressione booleana>) <blocco istruzioni>`
L'istruzione di selezione è utilizzata per selezionare se eseguire o no il blocco d'istruzioni associato, rispettivamente se la valutazione della espressione booleana abbia esito vero o falso.

Istruzione Else

Sintassi `else <blocco istruzioni>`.
L'istruzione else permette di eseguire il blocco d'istruzioni associato quando l'espressione di selezione precedente sia risultata false.
Una istruzione else è usabile solo direttamente successivamente a una istruzione di selezione.

Istruzione Else if

`else if(<espressione booleana>) <blocco istruzioni>`
L'istruzione else if equivale all'unione di un istruzione else e un Istruzione if poiché eseguirà il blocco d'istruzioni associato solamente se l'espressione booleana della istruzione precedente ha avuto esito falso e se il risultato della espressione booleana dell'istruzione

else if ha esito vero.
Come per l'istruzione else è possibile definirla solo direttamente successivamente a un Istruzione di selezione, è possibile concatenare molteplici else if poiché considerata anch'essa come istruzione di selezione.

Istruzione while

```
while(<boolena>) <blocco istruzioni>
```
L'istruzione while fa parte della categoria delle istruzioni di ciclo, viene usata per ripetere il blocco d'istruzioni associato fino a che l'espressione booleana, valutata precedentemente a ogni esecuzione del blocco, non risulti falsa.

Istruzione for

```
for([<init>]; [<controllo>]; [<post>]) <istruzioni>
```
L'istruzione for è un altro modo per creare cicli, utilizzata principalmente come abbreviazione dell'istruzione while.
Si divide in tre parti principali:

1. Inizializzazione: è un'espressione che viene valutata una volta all'inizio del ciclo, é legale definire al posto di un espressione una istruzione di dichiarazione (la variabile dichiarata avrà come ambito il blocco dell'istruzione for);

2. Espressione di controllo: definisce se continuare a eseguire il ciclo se risulta vera o terminarlo se risulta falsa.
   Se omessa verrà preso come default l'espressione letterale `true` creando un ciclo infinito;

3. Espressione post ciclo: è un'espressione che viene valutata dopo l'esecuzione del blocco d'istruzioni del ciclo ma prima della valutazione della espressione di controllo.

Istruzione continue

```
continue;
```
L'istruzione continue avvia una nuova iterazione dell'istruzione di ciclo più vicina che la racchiude.

Istruzione break

```
break;
```
L'istruzione break termina l'istruzione di ciclo più vicina che la racchiude.

Istruzione di fine partita

```
winner <lista di lista di giocatori>;
```
L'istruzione di fine partita, detta anche di vittoria, termina l'esecuzione del gioco definendo le classifica finale come definita dal argomento passato: le posizioni sono definite come una lista di giocatori, se due o più giocatori si trovano a pari merito è sufficiente inserirli nella lista all'indice corrispondente alla loro posizione, dove l'indice zero rappresenta il primo posto.

Nel caso si abbia bisogno di un solo vincitore è possibile utilizzare una sintassi abbreviata:
`winner <giocatore>;`

Istruzione d'input

`[optional] [<mod>] input <tipo> <identificatore> [<filtri>];`
Ha lo stesso comportamento degli input di azioni e funzioni 3.3.3, la variabile creata dall'input ha visibilità dello scope in cui è definita.
Non sono ammessi modificatori `function` 3.5.

3.2.6 Direttive

Le direttive sono istruzioni particolari definite solamente all'inizio del file, esterne a tutti i blocchi.

Direttiva d'importazione

Le direttive d'importazione sono utilizzate per utilizzare il codice definito in un altro file, evitando dei possibili conflitti e permettendo di avere un progetto ben organizzato.
Per maggiori informazioni vedere 3.12

Direttiva di gioco

`game "<nome␣del␣gioco>";`
La direttiva di gioco ha il compito di definire il nome del progetto e il file principale che deve contenere il tag `setup`; é utilizzabile solamente all'inizio del file come primo elemento.

Direttiva giocatori

`players <numero di giocatori>;`
La direttiva giocatori definisce il numero di giocatori, come *number* con parte decimale troncata, che saranno considerati per il gioco, definibile unicamente nel file contenente la direttiva di gioco.

3.3 Classi

`<visibilita> class <identificatore> <blocco classe>`
Una classe é utilizzata per definire un insieme di dati correlati, detti attributi, e di comportamenti che li modificano.
All'interno del blocco associato, detto blocco della classe o class block, sono definibili solamente i seguenti tag:

- attribute
- function
- action

Nel blocco è possibile riferirsi all'istanza stessa dell'oggetto tramite la parola chiave `this`.
Una certa istanza di una classe è assegnabile a un giocatore tramite la chiamata alla funzione predefinita `assign(player <giocatore>)`.
I comportamenti specifici durante l'assegnamento sono approfonditi in 3.3.2

3.3.1 Attributi

Un attributo è una variabile associata alla definizione della classe in cui è dichiarato. Il valore in sé della variabile è legato all'oggetto ovvero all'istanza della classe.
Per definire un attributo si utilizza il tag `attribute` come nella seguente sintassi:

```
attribute <identificatore>
{
  returns <tipo>;
  value <espressione>;
}
```

Dove:

- `returns`: definisce il tipo dell'attributo
- `value`: definisce il valore di prima inizializzazione dell'attributo tramite l'utilizzo di un'espressione dello stesso tipo indicato nel `returns`

Essendo una sintassi prolissa, si offre un'alternativa sintattica tramite l'utilizzo delle istruzioni di dichiarazione e inizializzazione.

```
<tipo> <identificatore> = <default>;
```

3.3.2 Visibilità

La visibilità di una classe definisce il suo comportamento e l'assegnamento rispetto al giocatore.

Visibilità locale

La visibilità locale o `local` è considerata il default, ovvero utilizzata se non è specificato altrimenti, possono esistere molteplici istanze di una classe locale create tramite l'espressione d'istanziazione.
In generale le classi locale servono per definire i ruoli, con le loro azioni, che i giocatori possono intraprendere nel corso della partita.
Quando si assegna un'Istanza di una classe locale a un giocatore, il giocatore precedente perde la proprietà dell'istanza concedendola al giocatore assegnato.

Visibilità di gruppo

La visibilità di gruppo o `group` è utilizzata per specificare delle classi le cui istanze possono essere condivise tra molteplici giocatori, che possono accedere alle azioni fornite dalla classe come se fosse di loro proprietà.
Come per le classi a visibilità locale, si possono creare molteplici istanze tramite l'espressione d'istanziazione.

Il principale utilizzo di una classe di gruppo è creare dei gruppi di giocatori che abbiano azioni e attributi comuni.
Quando si assegna un giocatore a una istanza di una classe di gruppo non avviene nessun trasferimento di proprietà, ma si aggiunge all'insieme di giocatori che la condividono.

Visibilità globale

La visibilità globale o `global` è utilizzata per specificare delle classi che sono limitate a una sola istanza in tutto il programma.
Tutti i giocatori possono accedere alle azioni delle classi globali.
L'istanza è accessibile in qualsiasi blocco che accetti delle espressioni di accesso utilizzando l'identificatore della classe come se fosse l'identificatore della variabile (meccanismo di accesso simile alle classi statiche di altri linguaggi).

3.3.3 Azioni

Le azioni rappresentano cosa possono fare i giocatori durante il corso del gioco, possono essere avviate automaticamente all'avvenire di certi eventi oppure invocate tramite un espressione di chiamata.
Un'azione è definita all'interno di una classe tramite il relativo tag:

```
action <identificatore> <blocco azione>
```

Argomenti dell'azione

```
[optional] [<mod>] input <tipo> <identificatore> [<filtri>]
```

Gli argomenti o input di un'azione sono dei tag speciali che servono per definire delle variabili inizializzate in base alla tipologia degli argomenti e utilizzate all'interno dei blocchi effetto di un'azione.
La tipologia dell'input è definita dal suo modificatore (`mod`) che ne influenza il comportamento, i due principali sono:

Modificatore giocatore `player [<espressione giocatore>]`
Detto anche modificatore player, è il default in caso di omissione del modificatore nella dichiarazione di un input.
Un input con il modificatore giocatore chiede direttamente al giocatore bersaglio di selezionare i valori con cui popolare l'input.
Se non specificato, il giocatore bersaglio di default dipende dalla visibilità della classe:

- Locale: il giocatore bersaglio è il proprietario della classe
- Globale: il giocatore bersaglio è il giocatore attivo
- Gruppo, si hanno due casi:
  - Se il giocatore attivo fa parte del gruppo sarà utilizzato come giocatore bersaglio
  - Se il giocatore attivo non fa parte del gruppo, un giocatore a caso sarà utilizzato come giocatore bersaglio

Modificatore auto `auto` Il modificatore auto o automatico non necessita di nessun input, il programma ha la responsabilità di filtrare tutte le istanze accettabili popolando la variabile definita dall'input.
Possono presentarsi dei fallimenti inattesi nel caso l'input non sia opzionale, non si tratti di una lista e nessuna istanza soddisfi i filtri.

Opzionale La parola chiave opzionale è un modificatore opzionale che permette alla variabile definita dall'input di assumere valore nullo quando l'input inizializza la variabile.
Gli input non opzionali garantiscono che forniranno dei valori non nulli oppure l'azione non sarà eseguita.

Filtri degli input

Un filtro di un input ha il compito di ridurre le possibili istanze selezionabili dall'input da tutte le istanze possibili del tipo dell'input.
Si definiscono secondo la seguente sintassi:

```
[optional] [<modificatore>] input <tipo> <identificatore>
{
  filter [<identificatore>] <blocco istruzioni>
  filter [<identificatore>] <blocco istruzioni>
  ...
}
```

È possibile semplificare ulteriormente la definizione se si ha bisogno di un solo filtro, omettendo il tag `filter`.

```
[optional] [<modificatore>] input <tipo> <identificatore>
{
  <istruzioni>
}
```

I filtri sono eseguiti su ogni oggetto tracciato di quel tipo, l'oggetto corrente è utilizzabile all'interno del blocco d'istruzioni tramite:

- l'identificatore dell'input nel caso sia omesso l'identificatore del filtro
- l'identificatore del filtro nel caso sia specificato

Il tipo della variabile dell'oggetto corrente dipende dal tipo richiesto nell'input:

- Se l'input non è una lista allora il tipo del filtro sarà lo stesso dell'input
- Se l'input è una lista allora il tipo del filtro sarà il tipo degli oggetti della lista.
  Per accedere al singolo valore in filtro in questo caso è necessario definire il filtro esplicitamente.

L'oggetto in esame sarà scartato o accettato in base al risultato booleano calcolato e restituito tramite istruzione di ritorno (il blocco di un filtro necessita di una istruzione di ritorno per essere valido).

Riutilizzo dei filtri Per riutilizzare i filtri il modo consigliato è racchiudere la logica di decisione dentro una funzione che viene chiamata da dentro i filtri.

Effetto

```
effect <blocco istruzioni>
```

Il tag effetto identifica il blocco d'istruzioni che viene avviato quando i triggers della azione sono soddisfatti.
In un blocco effetto non è possibile definire l'istruzione di ritorno poiché l'azione è trattata come una funzione senza tipo di ritorno, inoltre è possibile definire solo un tag effetto all'interno di una stessa azione.

Requisiti

```
require <blocco istruzioni>
```

I tag requisito di un'azione vengono utilizzati per disabilitare la possibilità di avviare l'azione tramite i triggers o istruzione di chiamata (nel caso l'azione sia disabilitata la chiamata non fallirà ma non avrà nessun effetto).
I blocchi requisito come i blocchi filtro necessitano di un valore di ritorno booleano che indica se l'azione è attiva o disabilitata, se sono presenti più di un tag requisito il risultato finale è composto da l'AND logico tra tutti i risultati dei blocchi requisito.

Triggers

```
trigger <evento> <blocco istruzione>
```

I trigger di un'azione sono i responsabili dell'avvio del flusso azione associato, quando l'evento che il trigger osserva viene lanciato, il blocco d'istruzioni viene eseguito; nel caso si abbia un esito positivo, ovvero un valore restituito `true`, l'esecuzione dell'azione sarà avviata.
Difatti il blocco d'istruzioni di un trigger, come i requisiti e blocchi filtro, necessita di un valore di ritorno booleano per poter valutare se eseguire o no l'azione.

Blocco trigger Ogni trigger osserva un particolare evento che è accessibile nel blocco tramite la parola chiave `event`, usata per creare delle logiche di attivazione più sofisticate riutilizzando gli stessi eventi.
In una specifica azione non è possibile definire molteplici triggers per il medesimo evento.

Priorità dell'azione L'architettura a eventi tramite i trigger introduce delle possibili condizioni di gara tra le varie azioni, per evitare delle esecuzioni in ordine inaspettato si introduce il concetto di priorità dell'azione.
La priorità di un'azione è un numero assegnato all'azione tramite la parola chiave `prio <numero>`, le azioni cui trigger sono stati valutati a vero saranno eseguite in ordine dalla priorità maggiore alla priorità minore.
Nel caso la priorità non sia stata definita oppure sia la stessa, si segue l'ordine di dichiarazione dell'azione.

3.4 Eventi

Gli eventi sono degli oggetti che sono lanciabili per avviare delle azioni che li osservano tramite i trigger.
Un evento è definibile tramite la seguente sintassi

```
event <identificatore> <blocco>
```

Il blocco di un evento ammette al suo interno solamente le dichiarazioni di attributi e di funzioni.
Quando un evento viene lanciato, i trigger che osservano quell'evento avviano immediatamente azioni corrispondenti, ma mettono in coda il loro avvio in base alla priorità dell'azione; questo implica una risoluzione dell'albero degli eventi/azioni in ampiezza (o Breadth First Search).

3.4.1 Eventi predefiniti

Gli eventi predefiniti sono degli eventi speciali che presentano dei comportamenti particolari rispetto allo standard, in generale servono per delle interazioni dirette con i giocatori.
Attualmente ne è definito solo uno:
L'evento `PlayerChoiceEvent` identifica un'azione che è attivabile da un giocatore durante il suo turno.
La struttura dell'evento è la seguente:

```
event PlayerChoiceEvent
{
  player active; // il giocatore che ha attivato l'azione
}
```

3.4.2 Funzioni degli eventi

Le funzioni generali degli eventi forniscono delle utilità globali generali:

lancia evento	`event.throw(event e)`	Lancia l'evento richiesto in modalità asincrona
lancia evento	`event.throw_sync(event e)`	Lancia l'evento richiesto in modalità sincrona

Sincronicità Un evento sincrono interrompe il flusso d'esecuzione del blocco corrente aspettando la risoluzione dell'albero degli eventi generato, al contrario un evento asincrono posticipa la risoluzione dell'evento dopo aver terminato l'esecuzione del blocco corrente.

3.5 Funzioni

Le funzioni sono utilizzate per raggruppare e riutilizzare facilmente un insieme d'istruzioni su uno specifico set d'input.
Sono definite tramite il tag

```
function <identificatore> <blocco funzione>
```

Argomenti

Gli input funzionano esattamente come gli argomenti delle azioni di una classe ma è possibile definire un'ulteriore tipologia d'input chiamata input della funzione, tramite il modificatore `function`.
Il modificatore obbliga a fornire tutti gli input che non possiedono un valore di default, quando si chiama la funzione.
Il modificatore function non permette la definizione dei filtri per il determinato input.

Effetto

È equivalente al blocco effetto di un'azione. In una funzione è possibile definire un solo blocco effetto.

Ritorno

Il tag di ritorno `returns [<type>];` é un tag obbligatorio speciale non associato a un blocco.
Definisce il tipo di ritorno della funzione, nel caso non si ritorni nessun tipo é possibile fornire il tag omettendo il tipo.

Chiamare una funzione

Per chiamare una funzione si utilizza l'espressione di chiamata dove l'identificatore corrisponde al nome della funzione da chiamare.
Per fornire alla funzione gli argomenti, ovvero gli input caratterizzati dal modificatore `function`, si utilizza la lista degli argomenti, in cui si indica l'identificatore dell'input da valorizzare, due punti e infine un espressione dello stesso tipo dell'argomento; per fornirne molteplici basta separarli con una virgola. Esempio:

```
hello(toPlayer: 1, msg: "hello world");
```

3.6 Interactables

`interactable <identificatore> <blocco>`
Un interagibile é una classe particolare specializzata per modellare i componenti che interagiscono direttamente con il giocatore, come le pedine, carte, ecc...
All'interno del blocco identificatore é possibile, come nelle classi, definire attributi, funzioni e azioni al suo interno.

Proprietario Il proprietario di un interactable é assegnato tramite la stessa sintassi delle classi, difatti tutte le nozioni sulla visibilità di una classe si applicano agli interactables.

Interazione con la board Gli interagibili possono essere piazzati nelle tile di una board, nel momento in cui sono piazzati in una tile, l'attributo `tile` viene valorizzatto con la tile in cui è stato messo.
Per muovere un interagibile a una tile si utilizza la funzione `move_to(tile t)` che sposta interagibile alla tile passata come argomento.

3.7 Mazzi

I mazzi sono delle classi specializzate che utilizzano la collezione `Stack`. Si definiscono come le classi tramite la keyword `stack`

```
stack <identificatore>
{
  returns <interagibile>;
}
```

All'interno di uno stack si possono definire attributi e funzioni, é necessario definire il tipo d'interagibile che è contenuto all'interno del mazzo.

3.7.1 Operazioni

Le operazioni effettuabili su un mazzo sono le stesse che possono essere eseguite su una collezione `Stack`.
Inoltre sono presenti le seguenti funzioni:

Rimescola	`<mazzo>.shuffle()`	rimescola il mazzo corrente ignorando gli scarti
Rimescola con scarti	`<mazzo>.reshuffle()`	Rimescola il mazzo compreso gli scarti

3.7.2 Scarti di un mazzo

Per rimuovere un interagibile dal gioco è necessario chiamare la funzione predefinita `<interactable>.discard(stack discard_stack)`, la funzione marca l'interagibile come scartato attraverso un flag predefinito `bool discarded` e lo aggiunge allo stack passato come argomento della funzione.
Ogni mazzo possiede uno stack predefinito chiamato `<stack>.discard_stack` che viene svuotato durante la chiamata `reshuffle()`.

3.7.3 Inizializzare un mazzo

Per creare la configurazione iniziale di un mazzo bisogna utilizzare in combinazione il tag default e le funzioni della collezione `Stack`, infatti il blocco default viene avviato dopo la creazione dell'istanza ma prima che sia restituita dall'espressione d'istanziazione.

```
default { <istruzioni per creare un mazzo di default> }
```

3.8 Giocatori

I giocatori sono rappresentati da un numero intero compreso tra zero e il numero di giocatori meno uno (estremi inclusi), questo permette di avere una corrispondenza uno a uno utilizzabile in espressioni matematiche per selezionare un altro giocatore da un certo di partenza (3.2.4).

3.8.1 Funzioni dei giocatori

Le funzioni generali dei giocatori forniscono delle utilità globali per recuperare informazioni su i players.

Tutti i giocatori	`player.all()`	Restituisce una lista contenente tutti i giocatori
Giocatore attivo	`player.active()`	Restituisce il giocatore attualmente attivo

3.9 Turni

Il turno definisce quale tra i giocatori viene definito attivo e il metodo di default per trasferire il controllo.
Il turno definibile piú semplice é il seguente:

```
turn <identificatore>
{
  default
  {
    // Passa il controllo al giocatore successivo
    // usando una espressione giocatore
    return turn.active() + 1;
  }
}
```

Si possono definire molteplici turni che sono attivabili in qualsiasi momento lo si desideri attraverso la funzione `turn.use(string turn, optional string phase)`, dove turn e phase sono rispettivamente l'identificatore del turno della fase prescelti.
Quando la funzione use è chiamata, il turno attivo è disattivato e sostituito da quello richiesto.

3.9.1 Definizione default del turno

Il turno di default, o tag di default del turno, é un blocco d'istruzioni di tipo player che viene chiamato quando la funzione senza parametri `turn.pass()` viene chiamata.
Definisce il passaggio di controllo da giocatore a giocatore e di fase in fase di default, per definirlo si utilizza l'omonimo tag:

```
turn <identificatore>
{
  phase <identificatore>
  {
    <inputs>
    effect { <istruzioni> }
  }
  default { <istruzioni> }
  // il tag di default deve ritornare
  // un oggetto non nullo di tipo player
}
```

Alcune funzioni molto utili per creare dei turni complessi si trovano tra le funzioni generali dei turni e dei giocatori, come `turn.phase.active()` e `player.active()`.

Fasi dei turni Un turno deve definire almeno una fase tramite il blocco `phase`, che rappresenta una azione speciale della classe non richiamabile dall'esterno come le azioni di una classe.
All'interno del blocco fase è legale definire un unico tag effetto e zero o piú tag input utilizzabili all'interno dell'effetto.
Le istruzioni contenute all'interno del blocco effetto definito nella fase sono eseguite immediatamente dopo la chiamata per il cambio di fase.

3.9.2 Funzioni dei turni

Le funzioni generali sono delle utilità che forniscono dei metodi per la gestione dei turni, come trovare chi é il giocatore attivo oppure passare alla prossima fase / giocatore in base alla implementazione del default.

Usa turno	`turn.use( string turn, optional string phase )`	Attiva il turno richiesto e, se specificata, in una precisa fase.
Turno attivo	`turn.active()`	Restituisce il turno attivo
Fase attiva	`turn.phase.active()`	Restituisce la fase del turno attiva
Passare al turno	`turn.pass( player p, optional string phase )`	Passa il controllo al giocatore richiesto e, se specificata, in una precisa fase
Passare il turno	`turn.pass()`	Passa il controllo al prossimo giocatore, valutato dal tag di default

3.10 Tabellone

I tabelloni sono delle classi specializzate, definite attraverso la parola chiave `board`, che rappresentano le tabelle di gioco, ovvero un oggetto con cui il giocatore puó interagire piazzando delle pedine, giocando carte, ecc...

```
board <identificatore> <blocco>
```

3.10.1 Tessere

Le tessere sono delle classi specializzate, definite attraverso la parola chiave `tile`, che rappresentano una singola tessera all'interno di un gruppo del tabellone.

```
tile <identificatore> <blocco attributi o funzioni>
```

All'interno del blocco di una tessera è possibile definire solamente delle funzioni o degli attributi.
Ogni tessera possiede degli attributi predefiniti:

Gruppo di appartenza	`<tessera>.group`	Riferimento al gruppo di appartenza della tessera
Connessioni	`<tessera>.connections`	Lista delle connessioni della tessera

3.10.2 Gruppi

I gruppi sono dei tag che permetteno di creare degli insiemi di tessere all'interno di una board, sono definiti tramite la seguente sintassi:

```
group <identificatore>
{
  geometry <geometria tessera>;
  grid
  {
    <matrice di tessere>
  }
}
```

Per accedere direttamente a un gruppo dal tabellone si utilizza la table dei gruppi dove la chiave equivale all'identificatore del gruppo e il valore è il gruppo stesso.

Matrice delle tessere

La matrice delle tessere si occupa di definire la struttura di default del gruppo.
All'interno del tag grid si definisce una riga come una lista d'identificatori tessera separati da una virgola e terminante con un punto e virgola.

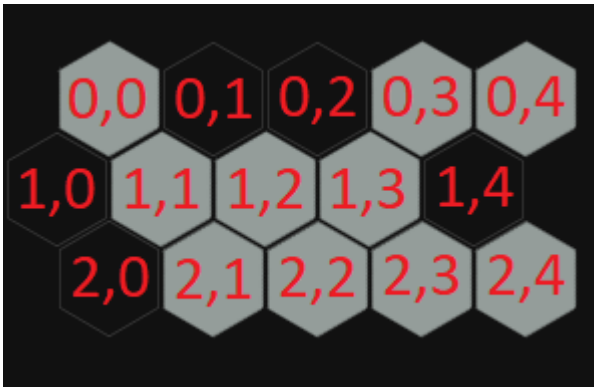
Geometria tessera

Esistono tre possibili geometrie supportate: quadrate (`square`), esagonali (`hex`), adiacenti (`graph`).
La geometria di una tessera all'interno del gruppo definisce come si definiscono all'interno del gruppo:

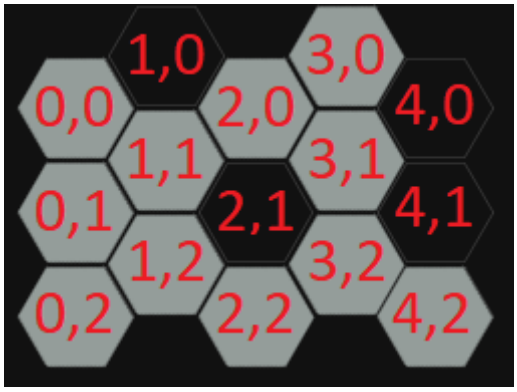
Geometria quadrata Non possiede nessun tratto particolare, il gruppo con questa geometria ha il suo punto di origine in alto a sinistra (0,0).

Geometria esagonale In questo caso é necessario definire un ulteriore tag body-less chiamato `orientation <orientazione>`, infatti la geometria esagonale possiede due possibili orientazioni, `row` (esagono a punta in alto) o `column` (esagono con faccia in alto); le orientazioni cambiano come la definizione del gruppo è interpretata:

- Nel caso dell'orientazione row: si hanno le righe pari spostate di una unitá verso destra
- Nel caso dell'orientazione col: si hanno le colonne pari spostate di una unitá verso il basso.



(a) Esempio coordinate con orientazione **row**



(b) Esempio coordinate con orientazione **col**

Geometria adiacente È utilizzata per poter rappresentare qualsiasi tipo di tabella creando un grafo bidirezionale, dove i nodi sono le tessere e le connessioni sono degli oggetti connessione.
Non é possibile utilizzare il tag `grid` per definire una geometria adiacente ma é necessario utilizzare il tag `default` accoppiato alla manipolazione manuale sia delle tessere che delle adiacenze di un gruppo.

Tessera bianca

Le geometrie quadrate ed esagonali possiedono la tessera bianca che rappresenta lo spazio vuoto, definita tramite la lettera `b` all'interno di un gruppo oppure tramite la tile `BlankTile`, se si utilizza al di fuori della definizione di un gruppo.

Recuperare delle specifiche tessere

Per accedere a una specifica tessera nei casi di geometrie regolari, si puó utilizzare la doppia indicizzazione del gruppo, trattandolo come se fosse una matrice di tessere.
I significati dei due indici cambiano in base alla geometria del gruppo:

- Quadrata: il primo indice è la colonna, il secondo è la riga
- Esagonale colonna: il primo indice è la colonna, il secondo é la riga
- Esagonale riga: il primo indice è la riga, il secondo è la colonna

Non è possibile accedere alle tessere di una geometria adiacente tramite indice poichè per definizione non possiede una struttura regolare facilmente mappabile uno a uno con una griglia.

Default

Il tag `default <blocco istruzioni>` viene eseguito appena dopo l'istanziazione del gruppo in cui è definito.
Permette la modifica non standard di un gruppo, conferendo maggior controllo sulle relazioni tra le tessere e i gruppi.

Connessioni tra tessere

La connesione tra due tessere viene modellata tramite la classe specializzata:

`connection <Identificatori> <blocco attributo o funzione>`

Possiede un attributo predefinito `tile to` che deve essere popolato con la tessera destinataria della connessione.

3.11 Ereditarietà

L'ereditarietà consente di definire delle classi derivate che ereditano o eseguono un override degli attributi, funzioni e azioni da una certa classe base

Ereditare da una classe base Significa avere accesso a tutte le funzionalità della classe padre come se appartenessero alla classe figlia, ovvero tramite l'operatore di auto riferimento `this`.

```
class base {}
class derived : base {}
```

Controllo sul tipo Un tipo derivato é anche dello stesso tipo di tutti i suoi tipi base, ovvero tutte le istanze della classe figlia restituiranno il valore `true` se confrontate con una classe antenata nella espressione di controllo del tipo.

3.11.1 Conversione del tipo

Un tipo derivato é convertibile a un tipo antenato senza dover utilizzare l'operatore di casting, questa operazione è chiamata "upcasting".
È importante sapere che le funzioni e le azioni chiamate dal tipo convertito avranno lo stesso comportamento che avrebbero se fossero chiamate su il tipo derivato.
Il caso contrario, il "downcasting" o conversione da tipo base a tipo derivato, è ammessa solamente se in origine l'istanza era del tipo derivato o figlia del tipo derivato.

Collezioni Le collezioni possono essere convertite ad altre collezioni il cui tipo che contengono segua comunque le regole di upcasting e downcasting sopra definite.

3.11.2 Override

Override permette di ridefinire il comportamento di una funzione o azione modificandone il tag effetto.
Per definire un override di una funzione é sufficiente definire nella classe derivante un'azione o funzione con lo stesso identificatore definito nella classe padre con la parola chiave `override`:

```
class Base
{
  function A {
    effect { // esegui istruzioni }
  }
```

```
}

class Derived : Base
{
  override function A {
    effect { // esegui altre istruzioni }
  }
}
```

Nell'effetto dell'override è possibile accedere agli stessi input dell'originale. Inoltre all'interno del blocco della classe derivata è possibile avviare la implementazione della base tramite il riferimento alla base `base`, che racchiude tutte le funzionalità definite nella classe padre.

3.12 Molteplici file

L'utilizzo di piú di un file é supportato attraverso l'utilizzo di speciali sintassi chiamate direttive d'importazione.

```
import "<percorso relativo al file>" [as <identificatore>];
```

Nella dichiarazione dell'importazione è possible assegnare un identificatore al file, attraverso il quale si accederà alle definizioni contenute all'interno di esso.
Nel caso sia omesso, tutte le definizioni all'interno del file importato non devono avere conflitti con gli identificatori del file in cui si importano.
Le direttive d'importazione sono definibili solamente all'esterno di qualsiasi blocco, prima di qualsiasi definizione o tag.

3.13 Punto di entrata

L'entry point è definito tramite il tag setup nello stesso file in cui si utilizza la direttiva di gioco. Dentro il blocco del tag si possono definire delle qualsiasi istruzioni che saranno eseguite prima di far partire il primo turno di gioco.

3.14 Esempio: Tris

```
// Nome del gioco descritto, definisce il file contenente
// il punto di entrata
game "tris";

// Numero di giocatori coinvolti nel gioco
players 2;

setup
{
  // Crea ed assegna il tabellone di default allo stato globale
  GameState.Board = new TrisBoard();

  // Crea ed assegna al giocatore 0 una classe PlayerState
  PlayerState firstState = new PlayerState()
  firstState.assign(0);

  // Crea ed assegna al giocatore 1 una classe PlayerState
  PlayerState secondState = new PlayerState()
  firstState.assign(1);

  // Dichiara il turno utilizzato
  turn.use("GameTurn");
  // Avvia il gioco passando il controllo al primo giocatore
  // in fase di Tick
  turn.pass(0, "TickPhase");
}

global class GameState
{
  TrisBoard Board = null;
  number TickedCount = 0;
}

function WinCheck
{
  function input TickTile a;
  function input TickTile b;
  function input TickTile c;
  returns PlayerState
  effect
  {
    // Controllo sulla riga
    if( a.AssignedTo != null &&
      a.AssignedTo == b.AssignedTo &&
      b.AssignedTo == c.AssignedTo)
    {
      return a.AssignedTo;
    }

    return null;
  }
}

turn GameTurn
{
  phase TickPhase { }
  phase WinCheckPhase
  {
    // prende tutte le tile della board
    Table<Table<TickTile>> tiles = {};
    for(number x = 0; x < 3; x += 1)
    {
      for(number y = 0; y < 3; y += 1)
      {
        tiles[x][y] = GameState.Board.TrisGrid[x][y] as TickTile;
      }
    }
```

```
      PlayerState winnerState = none;
      for(number i = 0; i < 3: i += 1)
      {
        // Controllo sulla riga
        winnerState = WinCheck(
          a: tiles[0][i],
          b: tiles[1][i],
          c: tiles[2][i]
        );
        if(winnerState != null) { break; }

        // Controllo sulla colonna
        winnerState = WinCheck(
          a: tiles[i][0],
          b: tiles[i][1],
          c: tiles[i][2]
        );
        if(winnerState != null) { break; }
      }

      // diagonal check
      if(winnerState == null)
      {
        winnerState = WinCheck(
          a: tiles[0][0],
          b: tiles[1][1],
          c: tiles[2][2]
        );
      }

      if(winnerState == null)
      {
        winnerState = WinCheck(
          a: tiles[2][0],
          b: tiles[1][1],
          c: tiles[0][2]
        );
      }

      // Se trovato un giocatore dichiaralo vincitore
      if(winnerState != null)
      {
        winner winnerState.player;
      }
      else if(GameState.TickedCount == 9)
      {
        // Crea la lista dei vincitori per imitare il pareggio
        List<List<player>> winners = { {0, 1} };
        winner winners;
      }
    }
    default
    {
      player active_player = player.active();
      string active_phase = turn.phase.active();

      if(active_phase == "WinCheckPhase")
      {
        // finita la fase di controllo della condizione di vittoria
        // continua con il prossimo turno
        turn.pass(active_player + 1, "TickPhase");
      }
      else
      {
        // fatto un segno passa
        // al controllo della condizione di vittoria
        turn.pass(active_player, "WinCheckPhase");
      }
    }
  }
```

```
tile TickTile
{
  PlayerState AssignedTo = null;
}

board TrisBoard
{
  group TrisGrid
  {
    geometry square;
    grid
    {
      TickTile, TickTile, TickTile;
      TickTile, TickTile, TickTile;
      TickTile, TickTile, TickTile;
    }
  }
}

class PlayerState
{
  action Tick
  {
    input TickTile chosenTile
    {
      return chosenTile.AssignedTo == null;
    }

    trigger PlayerChoiceEvent
    {
      return true;
    }

    effect
    {
      chosenTile.AssignedTo = this;
      GameState.TickedCount += 1;

      turn.pass();
    }
  }
}
```

3.15 Suggerimenti per l'implementazione

Il linguaggio è stato formulato come linguaggio tradotto.
In questa sezione vengono discusse delle possibili implementazioni supponendo come linguaggio bersaglio C#, principalmente i due passaggi di traduzione dal linguaggio TGPL a CSharp e l'utilizzo dell'assembly compilato dal codice generato.

3.15.1 Traduzione

I vari elementi del linguaggio sono tradotti ognuno in una equivalente struttura in CSharp, di seguito si possono trovare alcune delle conversioni meno triviali.

Listing 3.1: Codice in TGPL

```
class Test
{
  action TestAction
  {
    input player Classe classe { ... }
    trigger OnChange { }
    effect { classe.Call(); }
  }

  function TestFunction
  {
    function input number N;
    return number;
    effect { return N * 2; }
  }
}
```

Listing 3.2: Traduzione in CSharp

```
// dichiarata in un file apposito
public class OnChange : EventBase { }

public class Test
{
  public double test {get; set; } = 0;

  [EventTrigger(typeof(OnChange), 0)] // Type, priority
  [InjectedInput(typeof(Classe), "classe", ActionInputType.Player]
  public TGPLAction TestAction { get; }

  [FunctionInput(typeof(double), "N"]
  public TGPLFunction TestFunction()
  public double TestFunction(FunctionInput input) { get; }
}
```

Eventi

Gli eventi sono convertiti in una classe derivante da una classe base vuota utilizzata chiamata `EventBase`, la classe tradotta contiene tutti gli attributi sotto forma di proprietà publiche.

Funzioni

Le funzioni sono tradotte in una classe composta da due componenti, rispettivamente tradotti dal codice sorgente TGPL

```
public class TGPLFunction
{
  public Dictionary<string, Func<ActionInput, bool>> Inputs {get; set;}
  public Action<InjectedInput> Effect {get; set;}
}
```

Per aiutare l'esecuzione del codice ogni proprietà di tipo *TGPLFunction* è decorata con degli attributi specifici rappresentanti i metadati della funzione, come i vari input da popolare.
Effect é definito con argomento *InjectedInput*, una classe contenente una collezione chiave-valore dove la chiave è l'identificatore dell'input e il valore è l'oggetto inserito, popolato dall'*InputInjector*.
Questo meccanismo è condiviso nella traduzione di qualsiasi input, cambiando solo la modalità di acquisizione di esso dipendente dal modificatore definito.

Funzioni globali Le funzioni globali sono aggregate all'interno di una classe statica globale per fornire un accesso semplice al codice convertito.

Azioni

Le azioni sono tradotte in una classe composta da quattro componenti principali, rispettivamente tradotti dal codice sorgente TGPL:

```
// attributi di input e di triggers in ordine di definizione
public class TGPLAction
{
  public List<Func<bool>> Requires {get; set;}
  public List<Func<EventBase, bool>> Triggers {get; set;}
  public Dictionary<string, Func<ActionInput, bool>> Inputs {get; set;}
  public Action<InjectedInput> Effect {get; set;}
}
```

Per aiutare l'esecuzione del codice, come descritto in precedenza per *TGPLFunction*, ogni proprietà di tipo Action è decorata con degli attributi specifici rappresentanti i metadati dell'azione, come gli eventi di trigger e i vari input da popolare.

Modificatori delle classi I modificatori delle classi influenzano la traduzione:

- `local` viene tradotto con una proprietà Player associata rappresentante l'associazione al giocatore
- `group` viene tradotto con una proprietà lista di player per rappresentare il gruppo di giocatori associato
- `global` viene tradotta tramite la design pattern singleton, nello specifico possiede una proprietà statica che è l'istanza della classe

Classi

Le classi sono tradotte nell'equivalente in CSharp, dove gli attributi diventano proprietà publiche e le azioni e funzioni sono convertite secondo le speciche precedentemente descritte.

3.15.2 Utilizzo dell'assembly generato

Il secondo modulo principale del software di traduzione si occupa di avviare il codice tradotto, ovvero di estrarre tutte le classi utili per l'esecuzione del gioco, detto anche *ExecutionUnit* si compone di tre componenti principali:

- *CodeProcessor*
- *EventManager*
- *InputInjector*

InputInjector

L'*InputInjector* è il componente del processore con lo scopo di gestire la popolazione, secondo le specifiche del modificatore, degli input utilizzati nelle varie azioni e funzioni. Il comportamento di recupero di uno specifico input è definito dai valori del relativo attributo di metadati *InjectedInputAttribute*, in cui si definisce il tipo di oggetto, l'identificatore da utilizzare e la modalitá di recupero:

- Player: dopo aver effettuato i filtri definiti richiede a un giocatore di scegliere l'input tramite l'interfaccia utente
- Auto: dopo aver effettuato i filtri popola direttamente l'input se valido
- Function: il valore é fornito direttamente dalla chiamata

ReferenceRegistrar Il *ReferenceRegistrar* ha il compito di mantenere tutti i riferimenti e il loro numero creati durante l'esecuzione del gioco, è possibile concepirlo come un dizionario CSharp:

```
Dictionary<Type, List<(object, int)>>
```

ovvero una collezione che associa a un certo tipo la lista degli oggetti creati durante l'esecuzione e il numero di riferimenti visibili.
Il registrar é principalmente interrogato dal *InputInjector* durante la fase di recupero degli input di una azione o funzione, restituendo, in risposta alla richiesta del processore, tutte le istanze di un certo tipo per poterle filtrare secondo le specifiche dei filtri applicati a quell'input.

EventManager

L' *EventManager* si occupa di mantenere, per ogni tipo di evento definito nel gioco, le azioni che ne sono in ascolto e il trigger specifico dell'azione associato all'evento ascoltato. Una delle possibili implementazioni è un semplice dizionario:

```
Dictionary<Type, List<(Action, Func<EventBase, bool>)>>
```

interrogato dal processore per recuperare le azioni correlate a uno specifico evento generato all'interno di un blocco d'istruzioni. L'*EventManager* risponde alla richiesta del processore con la lista delle azioni e del trigger associato.
In questa semplice implementazione, a ogni istanziazione di una classe contenente azioni è necessario registrare nell'EventManager i triggers dell'azione. Ovviamente non è un sistema ottimizzato: un modo piú efficiente sarebbe creare dei metadati di ogni azione

definita all'interno della classe, che ne descriva i filtri utilizzati nel flusso di esecuzione delle azioni. I metadati conterrebbero non solo le informazioni per i triggers ma anche per i require e gli input dell'azione associata. Nondimeno, a scopo di mantenere semplice la implementazione suggerita e, di conseguenza, la sua descrizione, si é scelto di optare per la versione meno efficiente.

CodeProcessor

Il *CodeProcessor* é il responsabile di eseguire effettivamente le istruzioni contenute nei blocchi tradotti, durante l'esecuzione di un blocco se si presenta una generazione di eventi (sincrona o asincrona) la gestisce con le tecniche opportune:

- Sincrona: blocco dell'esecuzione fino a risoluzione dell'intero albero degli eventi
- Asincrona: posticipazione dell'esecuzione dell'albero degli eventi alla termine del blocco corrente

In entrambi le modalità contatta l'*EventManager* che gli restituisce la lista di tutte le azioni in ascolto di un certo evento, per ogni azione, contenuta all'interno della risposta del gestore degli eventi, il processore avvia il flusso azione:

Flusso azione Il flusso azione si divide in cinque passaggi principali:

1. Valutazione dei require, ovvero se l'azione è attivata oppure disattivata
2. Nel caso sia attivati si procede con la valutazione del trigger associato all'evento
3. Se positiva si recuperano gli input tramite l'*InputInjector*
4. Se non si sono presentate eccezioni si esegue effetto dell'azione
5. Infine se al suo interno sono presenti degli eventi generati si risolvono

Interfaccia utente Il processore utilizza un interfaccia utente dell'applicativo per mostrare informazioni di esecuzione o gioco necessarie al giocatore, come il tabellone.

Conclusione

Tabletop Game Prototype Language fornisce molti degli strumenti necessari a una protipizzazione agile, distanziandosi dal concetto di un gioco come macchina a stati finiti, riconducendo lo sviluppo a un'architettura a eventi, ovvero a una visione azione e reazione.
Durante la protipizzazione ne consegue una incrementata abilità comunicativa tra gli sviluppatori, progettisti e possibili esterni, grazie a una sintassi maggiormente esplicitativa a discapito di diventare prolissa.
Nonostante la maggior chiarezza, presenta dei punti di espansione non triviali:

- Gestione della concomitanza delle azioni di molteplici giocatori;
- Un sistema di gestione del tabellone meno ingrombante nel programma e maggiormente permissiva;
- Gestione delle espansioni, ovvero aggiunte o modifica a sistemi stabiliti nel gioco principale.

Inoltre, rispetto alla prototipizzazione in versione cartacea presenta svantaggi e vantaggi, principalmente il tempo d'implementazione di una nuova funzionalitá o modifica di una esistente é piú lenta, ma in opposizione permette una condivisione piú efficiente tramite i vari sistemi di condivisione digitale.

Bibliografia

[1] Google DeepMind. *AlphaZero: Shedding new light on chess, shogi, and Go*. URL: https://www.deepmind.com/blog/alphazero-shedding-new-light-on-chess-shogi-and-go. Data di riferimento: 14/12/2021.

[2] Michael Genesereth e Nathaniel Love. ≪General Game Playing: Game Description Language Specification≫. In: (dic. 2022).

[3] Roberto Ierusalimschy. *Programming in Lua - Tables*. URL: https://www.lua.org/pil/2.5.html. (Data di riferimento: 09/12/2022).

[4] Microsoft. *C# Documentation*. URL: https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/. (Data di riferimento: 10/12/2022).

[5] Michael Thielscher. *A General Game Description Language for Incomplete Information Games*. URL: https://cgi.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf. Data di riferimento: 14/12/2022.

[6] Michael Thielscher. *GDL-III: A Description Language for Epistemic General Game Playing*. URL: https://www.ijcai.org/proceedings/2017/0177.pdf. Data di riferimento: 15/12/2022.