Progetto di Programmazione 2019

`ASCII Maze`

v1.2

07/01/2020

Changelog

v1.2
- Nuova sezione 5.1 sulle alternative al costrutto `do` per inizializzare l'oggetto `maze`
- Correzioni refusi

v1.1
- Nuova sezione 5.3 sugli Array2D

v1.0
- Prima versione

Indice

1 REGOLAMENTO

Il progetto consiste nella realizzazione di un semplice videogioco in stile ASCII ART. In particolare, viene richiesto di sviluppare un programma che permetta la creazione di labirinti *(maze)* dove poter giocare in modalità interattiva o in modalità automatica.

1.1 Indicazioni generali

- I gruppi possono essere composti da al più 3 studenti.
- Leggere con attenzione le istruzioni di consegna che trovate al paragrafo 1.5. Progetti **consegnati in modo errato o fuori specifiche** verranno considerati **insufficienti**.
- È consentita la modifica della struttura fornita del progetto, previa discussione con il docente.

1.2 Plagio

I progetti verranno controllati al fine di verificarne l'originalità e la loro autenticità. Naturalmente, non è ostacolato il libero scambio di idee tra gruppi e l'aiuto reciproco purché il codice prodotto sia sviluppato in totale indipendenza tra i diversi gruppi. In caso di plagio, o altri comportamenti scorretti, l'esame di laboratorio verrà considerato **nullo** per tutti i componenti dei gruppi coinvolti, **tutti gli studenti coinvolti saranno ritenuti ugualmente responsabili**.

1.3 Requisiti di progetto

I requisiti minimi per questo progetto sono:

- implementazione dell'algoritmo di generazione di un labirinto casuale $w \times h$; è inoltre richiesta la sua corretta visualizzazione su schermo, mediante l'utilizzo del motore grafico messo a disposizione, oppure attraverso altri metodi concordati con il docente;
- sviluppo della modalità di gioco interattiva: un giocatore può dare comandi interattivi mediante l'utilizzo di un dispositivo di input data (es: tastiera) al fine di poter risolvere il labirinto generato. L'interazione deve essere facilmente visualizzabile per l'utente finale tramite la console;
- sviluppo della modalità automatica di risoluzione: il programma deve poter risolvere automaticamente il labirinto generato e mostrare graficamente la soluzione al giocatore.

L'utente utilizzatore del programma deve poter scegliere tra le diverse modalità di gioco, possibilmente tramite un menu oppure una qualche altra forma interattiva di interfaccia grafica.

1.4 Elementi di valutazione

Sebbene il progetto possa essere sviluppato in gruppo, la **valutazione finale** sarà comunque **individuale**. In particolare, il voto finale terrà in considerazione il risultato della prova orale e i seguenti criteri:

- realizzazione dei requisiti minimi [sezione 1.3] richiesti nel progetto: sviluppo dell'algoritmo di generazione, della modalità interattiva e della risoluzione automatica del labirinto.
- qualità del **codice**: oltre alla sua correttezza, saranno considerate l'eleganza della soluzione implementata, l'utilizzo di indentazione corretta e consistente, la suddivisione in sotto-funzioni ed altri aspetti di tipo qualitativo;

- qualità della **documentazione**: codice incomprensibile o **mal documentato** sarà valutato negativamente[1]

1.5 Indicazioni per la consegna

Il template del progetto è contenuto all'interno di un archivio zip disponibile all'interno della pagina `Moodle` del corso. La consegna del progetto deve essere tassativamente effettuata in formato zip su Moodle, con nome `Maze_[group_id].zip` dove `group_id` è l'identificatore del gruppo (consultabile su Moodle in fase di registrazione del gruppo, colonna "Gruppo" con formato "LabProg2019 ID"). Ricordiamo che è vostro dovere verificare che il contenuto dello zip consegnato sia corretto e contenga una *solution* di Visual Studio compilabile ed eseguibile.

1.6 Uso della libreria standard di F#

Al fine di non complicare inutilmente l'implementazione del progetto vi viene consentito - anzi, consigliato - di utilizzare le funzioni di libreria offerte dal linguaggio. Per maggiori informazioni vi consigliamo di leggere la documentazione ufficiale dei moduli *List*, *Array* e *String* di F#. In generale, cercando il nome di un modulo o di una funzione F# attraverso un motore di ricerca qulasisi sul web è facile trovarne la documentazione.

1.7 Modifiche al presente documento

È possibile che vengano applicate delle piccole modifiche al presente documento dopo la sua pubblicazione. Ogni versione del documento sarà associata ad uno specifico numero di versione indicato in prima pagina. Un *changelog* delle modifiche sarà pubblicato ad ogni aggiornamento.

2 INTRODUZIONE

In questa sezione definiremo il termine maze e vi daremo delle indicazioni riguardo gli algoritmi di creazione. Per quanto riguarda la generazione è possibile trovare un moltitudine di varianti e accortezze per la sua realizzazione. Se siete interessati all'argomento e volete cimentarvi in qualcosa di maggiore complessità potete implementare l'algoritmo di generazione che più vi aggrada. Per un esempio molto simpatico si veda il metodo proposto da Okamoto et al.[1] disponibile qui.

2.1 Definizione

Un *maze* si può definire come una rete confusa di percorsi intercomunicanti. In inglese esistono due parole per tradurre quello che in italiano viene genericamente chiamato labirinto e sono *maze* e *labyrinth*. La differenze fra un maze e un labyrinth sono principalmente strutturali [2], infatti un maze è traducibile in italiano come labirinto *multicursale*, mentre un labyrinth è un labirinto *unicursale*. La differenza fra un labirinto unicursale e multicursale sta nel fatto che nel primo il percorso, seppur lungo e complicato, non ha false piste, né possibilità di errore [2]. In questo progetto verrà chiesto di sviluppare la seconda struttura, ovvero quella multicursale, che chiameremo per semplicità maze. Due esempi che mettono a confronto i due tipi di labirinti sono proposti in Figura 1a e Figura 1b.

[1] Ricordiamo che non è necessario commentare ogni riga: è necessario commentare ciò che non è immediatamente lampante leggendo il codice, oppure gli algoritmi principali e le funzioni più importanti.

[2] A volte però, nell'uso comune, sono usati come sinonimi

2.2 Generazione di un maze

Sebbene possa sembrare solamente un semplice rompicapo, nel corso degli anni sono state proposte diverse soluzioni per la sua generazione, molti dei quali basati sulla teoria dei grafi. Qui di seguito, in Algorithm 1, vi proponiamo un semplice algoritmo ricorsivo per la loro generazione che ben si adatta ad essere implementato tramite l'uso di un linguaggio di programmazione funzionale.

Algorithm 1 Algoritmo di generazione di un maze

Funzione GENERAZIONE_MAZE(cella corrente)
  Segna la cella corrente come visitata
  Fino a quando hai celle adiacenti non ancora visitate
    $C_{adj} \leftarrow$ Scegli casualmente una di esse
    Rimuovi il muro fra la cella corrente e C_{adj}
    Invoca *generazione_maze* sulla cella C_{adj}

Potete trovare altri algoritmi e maggiori informazioni sugli algoritmi di generazione anche su Wikipedia alla pagina maze generation algorithm.

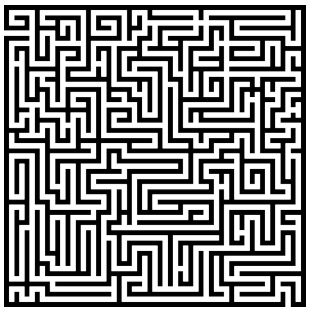
2.3 Motore grafico

Al fine di potervi facilitare l'interazione con la console, per potervi disegnare sopra in modo semplice ed intuitivo, è stato sviluppato un piccolo motore grafico 2D basato su grafica ASCII anziché normali pixel. Di seguito viene descritta la struttura generale del sistema, le modalità di interazione con lo stesso e degli esempi del suo utilizzo.
I tipi principale resi disponibili sono 4: `engine`, `pixel`, `image` e `sprite`.

2.3.1 Engine

Il tipo `engine` è composto principalmente da 4 proprietà: larghezza, altezza, il limite di frame per secondo (opzionale) e il numero di buffer da utilizzare (opzionale). Larghezza e altezza rappresentano rispettivamente la larghezza e l'altezza della console da visualizzare. Il limite di frame per secondo (fps) indica la frequenza di aggiornamento massima della schermata simulata con la console. Infine, il numero di buffer da utilizzare definisce il comportamento della tecnica *multiple buffering*[3].

(a) Labirinto unicursale, immagine presa da Nordisk familjebok, Nils Linder et al., terza edizione, 1937.



(b) Labirinto multicursale o maze

Figura 1: Esempi di tipi diversi di labirinti

2.3.2 Pixel

Il tipo `pixel` rappresenta un pixel all'interno della console, ovvero un carattere con un determinato colore di *background* e di *foreground*. Il tipo `pixel` è un alias introdotto nel modulo `Gfx` del tipo `CharInfo` originalmente definito nel modulo `External`.

2.3.3 Image

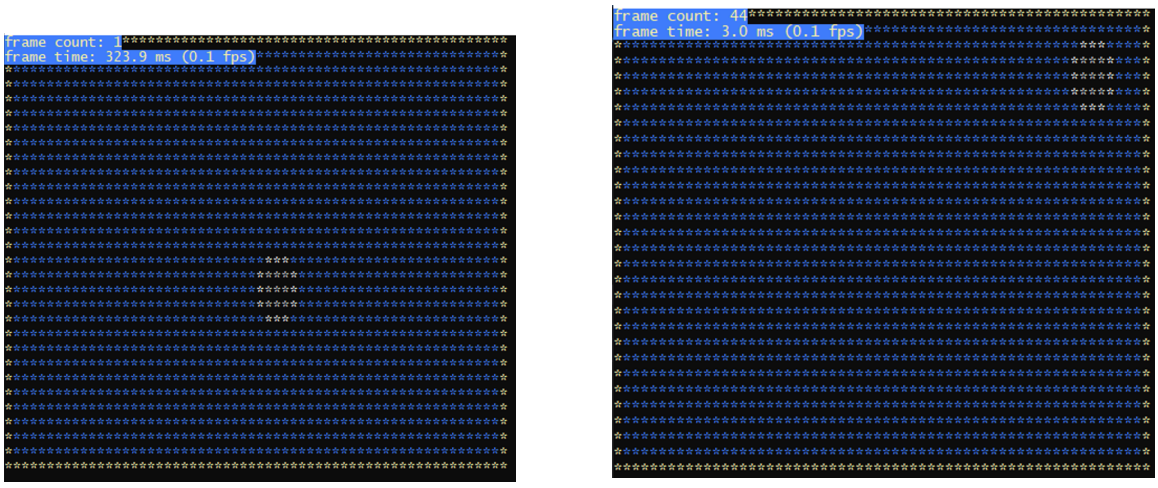
Il tipo `image` rappresenta un'immagine all'interno del nostro motore ed è composto principalmente da 3 proprietà: larghezza, altezza e array [4] di pixel. Ad esempio, un maze può essere rappresentato come una *image* dove i muri sono identificati da un carattere ASCII, come `'|'`, e l'assenza di muro può essere rappresentata con il carattere ASCII `' '`.

2.3.4 Sprite

Il tipo `sprite` è semplicemente un'immagine con 3 coordinate che rappresentano la sua posizione all'interno della console, che sono rispettivamente x, y e z; dove x rappresenta la coordinata sull'asse orizzontale, la y la coordinata sull'asse verticale e la z identifica la profondità dello sprite sulla console. L'origine dello schermo, ovvero la posizione (0,0,z) corrisponde all'angolo in altro a sinistra della console. Essendo il motore pensato principalmente per la visualizzazione in grafica 2D la terza dimensione z serve essenzialmente per rappresentare l'ordine con cui verrà visualizzato lo sprite sullo schermo. Di conseguenza, due sprite ($sprite_1$, $sprite_2$) aventi come coordinata z, rispettivamente z_1 e z_2, se $z_1 < z_2$, lo $sprite_1$ verrà stampato sulla console prima dello $sprite_2$, con la possibilità che i due sprite siano sovrapposti.

3 USO DEL MOTORE GRAFICO

In questa sezione viene mostrato brevemente un piccolo esempio fornito all'interno dello zip fornito per il task 0, con l'intento di realizzare una semplice demo interattiva come quella rappresentato in Figura 2.



(a) Stato iniziale

(b) Stato dopo il movimento del player verso l'angolo in alto a destre

Figura 2: *Screenshot* della demo2 fornita all'interno dello zip consegnato per il task 0.

Prendiamo in esame il sorgente *Demo2.fs* per capire com'è strutturato e per capire come strutturare un altro *main* simile o più complesso.

L'intestazione del sorgente contiene il nome del modulo che esso rappresenta; di seguito ci sono gli `open` che importano altri moduli dello stesso programma oppure moduli esterni di libreria.

```
module LabProg2019.Demo2

open System
open Engine
open Gfx
```

In questo caso `System` è un modulo della standard library di F#, mentre `Engine` e `Gfx` sono altri moduli del nostro programma.

Per prima cosa c'è la definizione del tipo `state` da passare al loop del motore. In questa demo abbiamo definito lo stato del gioco come un record [5] al cui interno è presente uno sprite a cui è stato dato il nome `player`.

```
type state = {
    player : sprite
}
```

Appena sotto ci sono due costanti globali[3]: la larghezza e all'altezza dello schermo che verranno passati al costruttore di `engine` ed, in generale, usate in questo modulo.

```
let W = 60
let H = 30
```

Perché definiamo due costanti anziché scrivere direttamente i numeri 60 e 30 dove serve? Perché in questo modo se vogliamo cambiare questi valori è sufficiente modificare il let della costante e non cambiare le varie occorrenze, potenzialmente multiple, sparse per il sorgente. E' una questione di ordine e di scalabilità del codice.

Più sotto c'è una funzione di nome `main`, che in realtà non è la vera entry-point del programma, ma viene chiamata dal vero main in *Main.fs*. All'interno della funzione `main`, viene definita la logica del vostro programma. Nell'esempio proposto, per prima cosa viene creato un oggetto di tipo `engine`:

```
let engine = new engine (W, H)
```

Successivamente viene definita la funzione `my_update`, avente firma `ConsoleKeyInfo -> wronly_raster -> state -> state * bool`, che più in basso viene passata al metodo `loop_on_key` dell'engine. Questa è la funzione che il motore chiama ad ogni fotogramma, perciò è qui che implementiamo ciò che il gioco (o la demo, in questo esempio) deve fare. In questo caso la funzione controlla semplicemente il tasto che viene premuto attraverso un semplice pattern matching e decide lo spostamento dello sprite `player` di conseguenza.

```
let my_update (key : ConsoleKeyInfo) (screen : wronly_raster) (st :
    state) =
     // move player
     let dx, dy =
         match key.KeyChar with
         | 'w' -> 0., -1.
         | 's' -> 0., 1.
         | 'a' -> -1., 0.
```

[3] In questo caso per *globali* intendiamo globali a livello di questo modulo, non globali per l'intero programma.

```
        | 'd' -> 1., 0.
        | _   -> 0., 0.
    st.player.move_by (dx, dy)
    st, key.KeyChar = 'q'
```

La funzione di update (che in questa demo abbiamo chiamato `my_update`) va passata all'engine e prende 3 argomenti: il tasto premuto (key), lo schermo dove disegnare (screen) e lo stato (st). Questo è un punto cruciale di questo motore: il motore è essenzialmente un loop che chiama ad ogni fotogramma la nostra funzione di update, passandoci tutto ciò che ci serve - il tasto premuto, il raster dove disegnare e lo stato. Maggiori informazioni sui raster e sugli oggetti in generale li trovate in sezione 5.

Si badi che il parametro screen non viene usato in questa demo: il motivo è che in questo caso non abbiamo bisogno di disegnare nulla oltre allo sprite del player, il quale viene disegnato automaticamente dal motore in quanto sprite registrato con la `create_and_register_sprite` (più sotto maggiori dettagli su questo aspetto). Tuttavia, in generale, un programmatore potrebbe voler disegnare altri elementi grafici che non sono sprite, quindi il motore passa anche il raster di output tra gli argomenti della update. E' importante capire una cosa: questo motore è *generale*, non è fatto solamente per questa demo o per questo gioco. Con questo motore è possibile fare giochi qualunque, quindi è pensato per casi molto variegati ed esigenze diverse: questa demo non ha bisogno di disegnare nulla oltre agli sprite automatici, quindi apparentemente sembra una complicazione in più avere il parametro screen; ma altri giochi o altre applicazioni di questo motore potrebbero voler fare qualcosa di diverso ed in tal caso avere a disposizione il raster di output è importante.

3.1 Stato

Lo stato merita un paragrafo a parte: il record di tipo `state` definito poco sopra rappresenta lo stato che desideriamo che il motore ci passi ad ogni chiamata della update. Potete vedere la update quindi come una funzione che - oltre al key ed allo screen - prende uno stato e produce un nuovo stato ad ogni fotogramma. Anche questo è un meccanismo molto generale - è una scelta di design del motore: essa permette al programmatore di definire stati qualunque, purché la funzione update ne prenda uno e lo ritorni aggiornato per il fotogramma successivo; ed così via.

Nella nostra demo lo stato è un record (spiegato in sezione 3) che contiene solamente lo sprite del player. La funzione `my_update` restituisce una coppia: il nuovo stato ed un booleano che indica al motore se uscire oppure no. Nel nostro caso lo stato non cambia mai, quindi viene restituito lo stesso stato `st` preso come argomento; il booleano invece produce `true` se l'utente ha premuto il tasto `'q'`, altrimenti `false`.

E' importante fare una riflessione sulla mutabilità dello stato. Lo stato contiene solamente lo sprite del player, che viene mosso dal metodo `move_by`, quindi come mai stiamo dicendo che lo stato resta immutato e che restituiamo lo stesso che abbiamo presto come argomento? Prima di dare una risposta facciamo un esempio ancora più semplice:

```
let simple_update (key : ConsoleKeyInfo) (screen : wronly_raster) (st :
    state) =
    st.player.move_by (1, 0)
    st, false
```

Questa semplicissima funzione di update prende lo stato di tipo `state` (il tipo è sempre lo stesso record di cui sopra), muove lo sprite del player di 1 pixel in orizzotale soltanto e ritorna lo stato `st` appena preso come argomento. Apparentemente lo stato non cambia, eppure lo sprite del player all'interno è stato mosso.

Qui la questione si fa intricata: cosa significa immutabile allora? Il metodo `move_by` ricalcola la posizione dello sprite ma non produce un *nuovo* sprite: viene solamente modificata la sua posizione, cioè i suoi campi x ed y. Ma lo sprite è sempre lo stesso oggetto in memoria - non è cambiato! Ed anche il record con lo stato è sempre lo stesso: un record con un campo solo (player) che contiene sempre lo stesso sprite.

3.1.1 Mutabilità vs. immutabilità

C'è una sottile differenza quindi tra modificare un valore (o un oggetto) e produrre un nuovo valore. Per chiarire ulteriormente questo aspetto facciamo un esempio diverso da questa demo.

```
type mut_state = { mutable counter : int }
type immut_state = { counter : int }

let mut_update key screen (st : mut_state) =
    st.counter <- st.counter + 1
    st, false

let immut_update key screen (st : immut_state) =
    { counter = st.counter + 1}, false
```

Abbiamo definito 2 tipi diversi: uno è un record `mut_state` con un campo `mutable` di tipo `int`; l'altro è equivalente ma definito con un campo non mutable. Appena sotto 2 diverse update che producono lo stesso risultato ma in due modi diversi. La `mut_update` prende uno stato `st` di tipo `mut_state`, incrementa il campo `counter` *modificandolo tramite un assegnamento*. La sintassi `x <- e` assegna ad x il valore prodotto dall'espressione e. Questa operazione modifica x, qualunque cosa essa sia, purché sia marcata come `mutable`. Nel nostro caso x è il campo `st.counter`, che è stato appunto marcato come `mutable`.

Viceversa, nell'update sotto, di nome `immut_update`, lo stato è di tipo `immut_state`, il quale è un record normale senza campi mutable. Ciò significa che non possiamo usare l'operazione di assegnamento `<-` per modificare, ma dobbiamo produrre un *nuovo* record come risultato se vogliamo cambiare lo stato. La sotto-espressione `{ counter = st.counter + 1 }` fa proprio questo: crea un nuovo record di tipo `immut_state` e lo *inizializza* con il valore preso dal campo `counter` dello stato `st` passato come argomento e sommandoci 1.

Il risultato non cambia: entrambe le varianti hanno lo stesso obiettivo - incrementare un contatore ad ogni chiamata, ma lo fanno in modi molto diversi.

3.2 Inizializzazione

Dopo aver definito la funzione di update è necessario definire gli sprite che verranno utilizzati e la loro posizione iniziale nello schermo. Per far ciò è possibile utilizzare la funzione `create_and_register_sprite` che permette di creare uno sprite e registrarlo all'interno del nostro engine. La registrazione serve a chiedere al motore di visualizzare ad ogni aggiornamento un determinato sprite. La firma della funzione è: `image * int * int * int -> sprite`, in caso non vi servisse lo sprite ritornato, lo potete semplicemente ignorare con la funzione `ignore` [6].

Nel caso specifico sono state create due immagini con le funzioni `image.rectangle` e `image.circle`.

```
// create simple backgroud and player
ignore <| engine.create_and_register_sprite (image.rectangle (W, H,
    pixel.filled Color.Yellow, pixel.filled Color.Blue), 0, 0, 0)
let player = engine.create_and_register_sprite (image.circle (2,
   pixel.filled Color.White, pixel.filled Color.Gray), W / 2, H /
   2, 1)
```

Infine, viene inizializzato il record che contiene lo stato iniziale, ovvero lo stato con il `player` appena creato, e viene lanciato l'engine con la funzione `loop_on_key` che aggiorna la schermata solamente se è stato premuto un tasto.

```
// initialize state
let st0 = {
    player = player
    }
// start engine
engine.loop_on_key my_update st0
```

Il metodo `loop_on_key` di `engine` prende 2 argomenti: la funzione di update che vogliamo che il motore invochi ad ogni fotogramma (che avevamo chiamato `my_update` e definito più sopra) e lo stato iniziale (che abbiamo chiamato `st0` e definito appena sopra). Si badi che il metodo `loop_on_key` non ritorna: il programma salta dentro il metodo e ci rimane virtualmente per sempre, fino a quando il motore non esce dal loop.

4 MAZE

Come descritto precedentemente il progetto consiste nello sviluppo di un videogioco basato su labirinti. Nel primo task viene richiesto di sviluppare la generazione del labirinto, che dati in input due valori w, h restituisce un labirinto composto da h righe e w colonne. Allo scopo di separare la logica del gioco dalla logica del labirinto, abbiamo definito una classe `maze` che conterrà proprietà e comportamenti che caratterizzano un labirinto. Nel corso di questa sezione approfondiremo alcuni costrutti di F# utili a comprendere il tipo `maze` ed altre entità avanzate presenti nel codice. Per ulteriori approfondimenti legati all'utilizzo di elementi avanzati di F# si guardi la sezione 5.

4.1 Scelta delle strutture dati

Un ruolo fondamentale nella stesura di un buon programma viene giocata dalla scelta delle strutture dati da utilizzare. Un labirinto può essere naturalmente interpretato come una matrice [4], il cui contenuto ne definisci la struttura. Questa è solo una piccola considerazione, in quanto viene lasciata allo studente la completa libertà di adottare qualsiasi tipo di struttura dati ritenga più consona allo sviluppo del progetto. Di fatti, un'altra soluzione valida potrebbe essere quella di utilizzare un Array come se fosse una matrice.
Le strategie che possono essere adottate per sviluppare il progetto sono molteplici, di seguito ne vengono descritte due possibili (non uniche):

1. Il metodo `generate` costruisce la struttura del labirinto su una rappresentazione logica (es. un Array2D di booleani che identifica se una cella ha un muro oppure no). Dopodichè viene definito un ulteriore metodo per convertire la struttura logica in un struttura grafica disegnabile (es. Image).

2. La struttura del labirinto viene scritta direttamente su una Image, combinando programmazione logica e rappresentazione grafica all'interno del metodo `generate`.

4.2 La cella di un maze

Altro elemento essenziale nel design di un buon codice è la scelta dei tipi di dato da utilizzare. Un suggerimento che viene dato è quello di definirvi da voi un nuovo tipo di dato *cell* che rappresenti le informazioni di una singola cella del labirinto dal punto di vista logico, non

grafico. Questa scelta risulterò molto importante per lo sviluppo della vostra soluzione, essa di fatto potrebbe portare a semplificazioni oppure a complicazioni, nel caso di una scelta non del tutto consona al problema. Un esempio di tipo *cella* è il seguente, incluso il pretty-printer[4]:

```
[< Diagnostics.DebuggerDisplay("{ToString()}") >]
type cell () =
    member val topWall = true with get, set
    member val bottomWall = true with get, set
    member val leftWall = true with get, set
    member val rightWall = true with get, set
    member val visited = false with get, set

    override this.ToString () =
        let sb = new StringBuilder ()
        if this.topWall then sb.Append 'T' |> ignore
        if this.bottomWall then sb.Append 'B' |> ignore
        if this.leftWall then sb.Append 'L' |> ignore
        if this.rightWall then sb.Append 'R' |> ignore
        let s = sb.ToString ()
        if this.visited then s.ToUpper () else s.ToLower ()
```

Ogni membro è un attributo booleano che identifica la presenza o meno di un muro nei 4 possibili lati. Ovviamente questa non è l'unica, di fatti un'altra soluzione potrebbe essere la seguente:

```
type cell () =
    member val isWall = true with get, set
```

In questo caso ogni cella è identificata con un unico valore booleano che specifica la presenza o meno di un muro su quella cella.
Data la molteplicità di possibili soluzioni/design, viene dunque lasciata totale libertà allo studente di utilizzare una definizione da noi fornita oppure una vostra personale.

4.3 Esempi di design

Come descritto precedentemente, possono essere adottare molteplici design per strutturare il vostro codice e dare una forma logica al vostro labirinto. Di seguito ne vogliamo mostrare un semplice esempio, supponendo di voler codificare la struttura in Figura 3.

Alla rappresentazione reale del labirinto dobbiamo contrapporre la nostra struttura logica, ovvero la strutta e tipo dati che utilizziamo per rappresentarlo a livello di codice. Un esempio potrebbe essere quello di utilizzare una struttura Array2D (sopra descritta e fornita da F#) con tipo di dati *'char'*. A questo punto potremmo utilizzare il simbolo **'*'** per identificare un muro e con ' ' una cella valida. La matrice che codifica esattamente il labirinto in Figura 4 è di seguito illustrata:

5 ELEMENTI AVANZATI DI F#

In questa sezione spieghiamo alcune caratteristiche avanzante del linguaggio F# utilizzate in questo progetto. F# permette di definire tipi *nuovi* che rappresentano entità complesse. In

[4]Il metodo `ToString` converte l'oggetto in una stringa. L'annotazione `Diagnostics.DebuggerDisplay ("{ToString()}")` invece istruisce il debugger di Visual Studio ad invocare l'espressione `ToString()` per visualizzare i dati di questo tipo nel debugger.

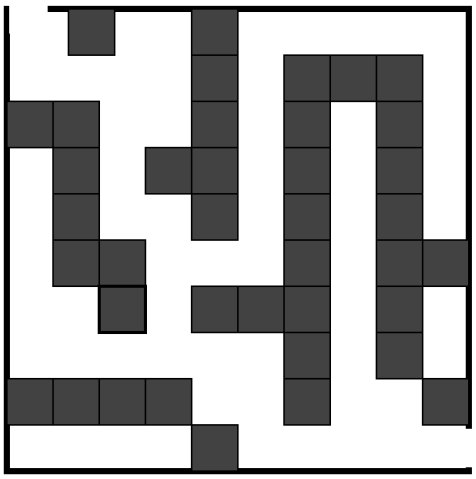


Figura 3: Esempio di algoritmo reale. I blocchi scuri rappresentano i muri, contrariamente i bianchi rappresentano percorsi validi per essere attraversati.

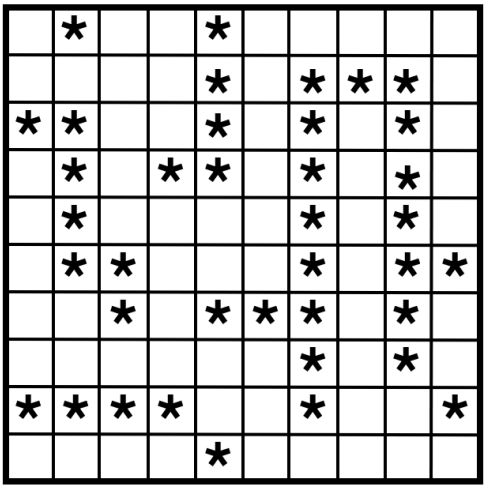


Figura 4: Codifica in Array2D con tipo di dato *'char'*.

generale in tutti i linguaggi di programmazione ad oggetti il programmatore è in grado di definire *classi* - anche in F# è possibile [7]. Le classi permettono al programmatore di definire dati e funzioni che operano su tali dati, sotto forma di un tipo nuovo. Una classe inoltre può essere *istanziata*, cioè è possibile *costruire* un *oggetto*, come un valore che appartiene a quella classe. Allo stesso modo in cui il numero 4, ad esempio, è un valore di tipo `int`, così la medesima relazione esiste tra oggetti e classi: un oggetto è un valore, una classe è un tipo.

Pur essendo un linguaggio funzionale, F# supporta la programmazione ad oggetti[5] e permette di definire nuovi tipi tramite il costrutto `class` e costruire oggetti tramite il costrutto `new`. In questo progetto vengono definite una serie di classi che rappresentano entità come il raster, un pixel, un labirinto ecc.: ogni entità è rappresentata da una classe che ne incarna il tipo; una classe da sola non è un valore però: per creare dei valori è necessario istanziare oggetti, pertanto, ad esempio, la classe `maze` non è un labirinto - è solamente il tipo dei labirinti.

Prendiamo in considerazione il tipo `maze`. All'interno del file Maze.fs viene già da noi abbozzato un tipo `maze` da completare:

[5]Così viene chiamato lo stile di programmazione basato su classi ed oggetti.

```
type maze (w, h) as this =
    // eventuali let locali con i dati interni dell'oggetto

    do this.generate    // object initializer

    member private __.generate =
        // qui implementate la generazione del maze

    // altri metodi possono essere aggiunti sotto
```

Nella prima riga si nota che per costruire un oggetto di tipo `maze` servono due parametri, w e h, che sono le dimensioni del labirinto e funzionano come due parametri di una funzione - una funzione speciale: il costruttore dell'oggetto. In un altro punto del codice, dove desideriamo ad esempio creare un oggetto di tipo `maze`, possiamo scrivere:

```
let my_maze = new maze (50, 30)
```

L'espressione `new maze (50, 30)` costruisce un oggetto di tipo `maze` invocando il suo costruttore con 2 argomenti: 50 e 30 per l'appunto. Dentro la classe `maze` tali argomenti si vedono col nome dei parametri w e h. Il let serve a mettere in una variabile (di nome `my_maze` in questo caso) l'oggetto appena costruito. Ora abbiamo davvero creato un labirinto.

5.1 Alternativa al costrutto `do`

Inizializzare un oggetto in F# tramite il costrutto `do`, come discusso appena sopra, può presentare dei problemi difficili da comprendere a fondo. In breve, ciò dipende dal modo in cui F# viene tradotto nel bytecode di .NET e dal comportamento a runtime della CLR - la virtual machine di .NET di Microsoft. Qualora dovessero presentarsi tali problemi, è naturalmente possibile fare in altro modo - come sempre ci sono più modi per scrivere il medesimo codice. Consigliamo ad esempio di togliere il costrutto `do` dalla classe, rendere pubblico il metodo `generate` e chiamarlo esplicitamente dopo aver costruito un maze. La classe `maze` diventa:

```
type maze (w, h) as this =
    // eventuali let locali con i dati interni dell'oggetto

    member __.generate =
        // qui implementate la generazione del maze

    // altri metodi possono essere aggiunti sotto
```

Ed il codice del chiamante:

```
let my_maze = new maze (50, 30)
my_maze.generate
// resto del codice
```

Possiamo inoltre vedere che viene definito un nuovo metodo [8] chiamato `generate`, il cui corpo è completamente vuoto. L'obiettivo del task 1 è quello di completare la definizione di questo metodo che deve costruire un labirinto $h \times w$, secondo l'algoritmo da voi scelto. Notare bene che il metodo `generate` viene chiamato in fase di inizializzazione dell'istanza. Quando viene creato un oggetto (tramite il costrutto `new`), viene anche eseguito il blocco che segue la keyword `do` - tale blocco si chiama *object initializer* e serve a fare qualcosa quando si viene costruiti. Nel nostro caso, quando qualcuno costruisce un oggetto di tipo `maze`, quello che vogliamo è invocare il metodo `generate` che lo popola generando il labirinto.

5.2 Altri esempi di classi e oggetti

Prendiamo per esempio la classe `sprite` definita nel modulo `Gfx`:

```
type sprite (img : image, x_ : int, y_ : int, z_ : int) =
    inherit image (img.width, img.height, img.pixels)

    member val x : float = float x_ with get, set
    member val y : float = float y_ with get, set
    member val z = z_ with get, set

    member this.move_by (dx, dy) =
        this.x <- this.x + dx
        this.y <- this.y + dy

    member this.move_by (dx, dy) = this.move_by (float dx, float dy)

    member spr.draw wr = spr.blit (wr, int spr.x, int spr.y)
```

Questo costrutto definisce una classe di nome `sprite`, sottoclasse di `image`, costruibile tramite 4 parametri: un oggetto di tipo `image` e 3 interi. Il tipo `image` è a sua volta una classe, definita sempre nel modulo `Gfx`. Non si confonda il fatto che `image` è superclasse di `sprite` col fatto che il primo parametro del costruttore di `sprite` è una `image`: sono due cose distinte. Quando una classe è sottoclasse di un'altra significa che la sottoclasse *eredita* tutti i membri della superclasse: quindi uno `sprite` può essere visto come una estensione di *image* [9]. Diverso invece è il fatto che per costruire uno `sprite` occorra invocare il costruttore passando 4 argomenti, di cui il primo è di tipo `image`.

La keyword `member` definisce delle funzioni specifiche per questa classe, dette *metodi* [8]. Le keyword `member val` definiscono campi espliciti con property automatiche [10].

Vediamo degli esempi di chiamata di metodi e property:

```
let img1 = new image (50, 30)   // crea una image
// disegna un rettangolo giallo
img1.draw_rectangle (0, 0, 50, 30, pixel.filled Color.Yellow)
let spr1 = new sprite (img1, 0, 0, 1)  // crea uno sprite con la image
spr1.x <- spr1.x + 1    // sposta la x dello sprite spr1 di 1 a destra
spr1.move_by (5, 6)     // sposta lo sprite spr1 di 5 in x e 6 in y
```

Il primo let costruisce un oggetto di tipo `image` invocando il suo costruttore con 2 parametri: viene costruita una immagine di 50 * 30 caratteri di default inizializzati a `pixel.empty` (pixel vuoto nero). La seconda linea non è un binding ma l'invocazione di un *metodo*, cioè una funzione definita come membro della classe: la sintassi `oggetto.metodo (argomenti)` è la sintassi di invocazione di un metodo.

I più attenti avranno però notato che il metodo `draw_rectangle` tuttavia non compare tra i metodi di `sprite` - com'è possibile invocare un metodo che non compare tra i membri? La risposta sta in un meccanismo chiamato *ereditarietà*: tramite la keyword `inherit` è possibile specificare una superclasse di questa classe, cioè un'altra classe dalla quale *ereditare* i membri. Questo significa che la classe `sprite` estende la classe `image`, pertanto tutti i membri di `image` sono inclusi anche in `sprite`.

L'implicazione di tutto questo è che, volendo, è possibile utilizzare uno `sprite` al posto di una `image`, ad esempio:

```
let img1 = new image (50, 30)          // crea una image
let spr1 = new sprite (img1, 0, 0, 1)  // crea uno sprite con la image
let spr2 = new sprite (spr1, 0, 0, 2)
```

L'ultimo let costruisce uno `sprite` e passando lo sprite `spr1` come primo argomento al costruttore anziché una `image`: ciò è legale in F# perché la classe `sprite` *estende* la classe `image`, quindi uno sprite è *anche* una image.

5.3 Uso di Array2D

Per rappresentare il labirinto sotto forma di un array bidimensionale dove ogni elemento è un oggetto di tipo `cell`, è necessario essere consapevoli di alcune sottigliezze. Riprendiamo in considerazione parte della definizione della classe `cell`:

```
type cell () =
    member val visited = false with get, set
    ...
```

Immaginiamo ora di voler creare un array bidimensionale di oggetti di tipo `cell`:

```
let a = Array2D.create w h (new cell ())
```

Il terzo argomento dell'applicazione rappresenta il valore con cui riempire l'array di dimensione $w*h$: tale valore è computato dall'espressione `new cell ()`, che è l'invocazione del costruttore della classe `cell`. Tuttavia provando poi a modificare un elemento qualsiasi dell'array ciò che accade è che vengono modificati *tutti* gli elementi dell'array. L'assegnamento del campo `visited` di una `cell` a qualsiasi coordinata in realtà modifica *tutte* le cell dell'array.

```
a.[0, 0].visited <- true
printf "%b" a.[1, 1].visited    // stampa true e non false!!
```

Questo accade perché il valore passato come terzo argomento alla `Array2D.create` è *un solo oggetto costruito una sola volta*, il cui indirizzo viene replicato per tutte le celle dell'array - solo l'indirizzo viene replicato, non l'oggetto, producendo quindi $w*h$ puntatori al medesimo oggetto in memoria!

Per ovviare a questo comportamento, dobbiamo fare in modo che ogni elemento dell'array abbia un oggetto di tipo `cell` *distinto*:

```
let a = Array2D.init w h (fun _ _ -> new cell ())
```

Usando la funzione `Array2D.init`, il terzo argomento, anziché essere il valore da replicare per riempire tutto l'array, è una lambda che viene *invocata ad ogni elemento* dell'array e a cui vengono passati riga e colonna come argomenti. In questo caso a noi le coordinate non servono: per ogni elemento dell'array ci basta semplicemente creare un oggetto nuovo di tipo `cell` ed ignorare le coordinate - ecco il perché dei 2 underscore sui parametri della lambda. Questo produce l'effetto desiderato: ora ogni cella dell'array è un oggetto diverso ed è possibile modificarlo singolarmente.

Riferimenti bibliografici

[1] Y. Okamoto and R. Uehara, "How to make a picturesque maze." in *CCCG*, 2009, pp. 137–140.

[2] P. Santarcangeli, *Il libro dei labirinti*, ser. Collana Rivelazioni. Sperling & Kupfer, 2000.

[3] Wikipedia, "Double buffering," [Apri link].

[4] Microsoft Documentation, "Array 2d," [Apri link].

[5] ——, "Records in f#," [Apri link].

[6] ——, "Ignore," [Apri link].

[7] ——, "Classes in f#," [Apri link].

[8] ——, "Methods in f#," [Apri link].

[9] ——, "Inheritance in f#," [Apri link].

[10] ——, "Member val in f#," [Apri link].