Manuale Utente

Università degli Studi di Bergamo

Laurea Magistrale di Ingegneria Informatica

A.A. 2022/2023

Corso di Linguaggi Formali e Compilatori (9 CFU)

cod. Corso 38070

Carne Federico – 1059865

Sommario

1 Introduzione

Benvenuto nel manuale utente di YARC (versione 1.0.0).

Il seguente documento è diviso in tre macro-sezioni:

- Una sezione introduttiva che spiega cos'è YARC.
- Una collezione di tutorial ed esempi su come usare YARC.
- Una sezione di riferimento in cui sono spiegati il linguaggio e il tool a riga di comando.

2 YARC

Prima di cominciare, cerchiamo di capire *cos'è* YARC e *cosa può fare*.

2.1 Introduzione

2.1.1 Cos'è YARC…

YARC (Yarc Ain't Replicator Composer) è un tool per la generazione di script eseguibili finalizzati alla generazione di dati sintetici.

Negli ultimi anni si è visto un grande sviluppo di tecniche di Machine Learning, anche grazie a una maggior quantità di dati disponibili e all'aumento della capacità computazionale dei computer. Ciononostante, acquisire dati per applicazioni custom e industriali è spesso un processo costoso, complesso e lungo, senza tener conto del tempo necessario per etichettare i dati. Una volta raccolti i dati per allenare un algoritmo, l'unico modo per averne di nuovi è effettuare nuovamente l'acquisizione sul campo.

Una delle tecniche per velocizzare la fase di raccolta dei dati è la Synthetic Data Generation (SDG): tramite l'utilizzo di algoritmi e simulatori, vengono creati artificialmente dati che rispecchiano la realtà. Nel campo di algoritmi di visione si possono usare engine di modellazione 3D e librerie sviluppate appositamente per comporre ambientazioni 3D e generare dati già etichettati. Questa soluzione porta a un processo di acquisizione molto più veloce, rapido e meno costoso.

YARC vuole fare da collettore per tutti gli engine e librerie presenti nel mondo della modellazione 3D, fornendo un'interfaccia uniforme e generale. YARC, infatti, fornisce agli utenti uno Scene Description Language (SDL) per la composizioni di scenari 3D e rende disponibili i tool per tradurre questi scenari in codice direttamente eseguibile all'interno di un engine 3D qualsiasi.

YARC segue il principio WORA (Write Once, Run Anywhere), se si vuole cambiare engine di generazione basta cambiare un solo parametro all'interno dello script.

2.1.2 … e cosa non è?

YARC non è un engine di modellazione 3D, con YARC non si possono costruire modelli 3D complessi e non dispone di un'interfaccia grafica per la composizione della scena. YARC si fonda sugli script, che sono a lato pratico file di testo.

Anche se tutto ciò può sorprendere qualche utente, è proprio questo che dà flessibilità e potenza a YARC: essere in grado di poter gestire ogni singolo aspetto della scena (anche se l'apprendimento è più complesso rispetto a un'interfaccia grafica).

2.1.3 Chi può usare YARC?

YARC è stato pensato principalmente per tutti coloro che hanno almeno una conoscenza base dei concetti di modellazione 3D, ma chiunque può approcciarsi a YARC.

È bene anche conoscere l'engine in cui si andranno a generare i dati, la maggior parte del lavoro per la generazione dei dati sintetici è a carico del software di modellazione, YARC semplifica il processo di specifica ma non può fare nulla per quanto riguarda performance, velocità o livello di realismo del render: l'utente deve sapere aggiustare le impostazioni dell'engine per raggiungere il risultato sperato.

3 Tutorial

In questa sezione vedremo come si installa YARC sul proprio dispositivo, come si usa e cosa può fare con una serie di esempi.

3.1 Installazione

Per installare YARC è necessario aver prima installato Python 3. Se non lo si ha già fatto, Python si può installare dal sito ufficiale.

Una volta installato Python, serve scaricare l'ultima versione del file `.whl` dalla repository ufficiale di YARC. I file `.whl` non sono altro che semplici file `zip` usati da Python e dal suo gestore di dipendenze per installare package e librerie di terze parti. Per installare YARC è necessario aprire un terminale e digitare il seguente comando:

```
$ pip install path/to/yarc-{version}-py3-none-any.whl
```

Una volta terminato, YARC sarà disponibile tra i comandi del terminale. Per verificare che l'installazione sia andata a buon fine basta digitare il seguente comando:

```
$ yarc --help
```

Se l'installazione è stata completata con successo, verrà mostrato a schermo l'help di YARC. Possiamo vedere che YARC mette a disposizione due comandi: `compile` e `translate-grammar`. Il primo comando si occupa di tradurre gli script YARC in codice eseguibile, il secondo è un comando di utility per gli sviluppatori e quindi per il momento lo ignoreremo.

Tutti gli esempi che seguono usano Omniverse come engine e la libreria Replicator.

3.2 Hello, World

Ora che YARC è stato installato possiamo creare il nostro primo script YARC:

```
scenario HelloWorld: Replicator

stage:
  cube = create Cube
  edit cube:
    translate to <0,0,0>
    semantics "class:cube"
```

(puoi copiare lo script e salvarlo in un file `.yrc`, non è necessario usare questa estensione ma aiuta a distinguere i tuoi script YARC dal resto dei tuoi file).

Per generare il file eseguibile basta aprire un terminale, portarsi nella stessa cartella dove abbiamo salvato lo script `hello_world.yrc` e digitare il seguente comando:

```
$ yarc compile hello_world.yrc
```

YARC processerà il file in ingresso e mostrerà a schermo il codice Python risultante. Se vogliamo salvare il risultato della traduzione possiamo usare l'opzione `-o`:

```
$ yarc compile hello_world.yrc -o hello_world.py
```

Ma cosa significa lo script precedente?

- La prima riga definisce l'header del file ed è obbligatorio. La keyword `scenario` introduce l'header ed è seguita dal nome dello scenario. Dopo il simbolo di `':'` vi è la libreria target, in questo caso Omniverse Replicator.
- Un file YARC è normalmente diviso in tre sezioni: una prima sezione di setting, la sezione principale che definisce lo stage e una sezione finale che contiene tutti i writer selezionati per i dati sintetici. La sezione di stage è obbligatoria e definisce la scena vera e propria. La sezione di stage è introdotta dalla keyword `stage`, dal simbolo`':'` e da un blocco indentato (chi ha familiarità con Python riconoscerà questa sintassi). Bisogna fare molta attenzione a indentare correttamente il codice.
- All'interno del blocco indentato creiamo un cubo e lo assegniamo alla variabile `cube`. Successivamente nel blocco di `edit` (indentato anch'esso) ne settiamo due attributi: la posizione nell'origine e la label semantica (`"class:cube"`).

3.3 Luci, camera, azione!

Lo script precedente è corretto ma così com'è non è molto utile, proviamo a espanderlo. Per prima cosa, aggiungiamo una luce e una camera nella sezione `stage`:

```
  light = create Light:
    intensity 6500
    temperature 5000
```

```
camera = create Camera:
  translate to <-100,0,0>
  look_at cube
```

Così facendo abbiamo create una luce con intensità pari a 6500 e temperatura pari a 5000K (per una lista degli attributi accettati da ogni oggetto, fare riferimento all'appendice) e una camera posizionata in (-100,0,0) e orientata in modo da guardare il nostro cubo.

Ora aggiungiamo della dinamica alla nostra scena: ogni frame andremo a spostare, ruotare e scalare il nostro cubo casualmente e ogni 5 frame a cambiare la temperatura della luce secondo una sequenza prefissata. Sempre nella sezione di `stage` aggiungiamo le seguenti istruzioni:

```
every frame:
  with cube:
    translate to Uniform(<-25,-25,-25>,<25,25,25>)
    rotate Uniform(<0,0,0>,<360,260,360>)
    scale Normal(<1,1,1>,<0.5,0.5,0.5>)

every 5 frame:
  temps = [4500, 5000, 5700, 6000, 6300]
  with light:
    temperature Sequence(temps)
```

Tutte le istruzioni definite all'interno del primo blocco `every` saranno eseguite ad ogni frame di generazione mentre le istruzioni nel secondo blocco saranno eseguite ogni 5. Questi vengono chiamati *trigger*. All'interno di un blocco `every` si possono utilizzare dei blocchi di `edit` per modificare gli attributi e il valore è estratto dinamicamente da una distribuzione (`Uniform` e `Normal`) oppure da una sequenza di valori (`Sequence`).

I più attenti avranno notato che si possono definire variabili e assegnargli un valore (in modo molto simile a Python, per chi conoscesse il linguaggio di programmazione). Queste possono essere definite ovunque nella sezione di `stage` ma è comodo definirle vicino a dove vengono utilizzate. Tutte le variabili, prima di essere utilizzate vanno dichiarate (altrimenti si genera un errore), una variabile può essere utilizzata all'interno del blocco in cui è stata definita o nei sotto-blocchi.

Ora che abbiamo la dinamica, l'unica componente mancante è la cattura dei frame generati. Per fare questo andiamo a definire la sezione `writers` subito sotto quella di `stage` e qui dentro inizializziamo un `BasicWriter` per catturare i frame RGB, le

maschere di segmentazione e le bounding boxes 2D (il `BasicWriter` è un writer specifico di Replicator)

```
writers:
  BasicWriter:
    rgb = true
    semantic_segmentation = true
```

Lo script completo risulta essere:

```
scenario HelloWorld: Replicator

stage:
  cube = create Cube
  edit cube:
    translate to <0,0,0>
    semantics "class:cube"

  light = create Light:
    intensity 6500
    temperature 5000

  camera = create Camera:
    translate to <-100,0,0>
    look_at cube

  every frame:
    with cube:
      translate to Uniform(<-25,-25,0>,<25,25,0>)
      rotate Uniform(<0,0,0>,<360,260,360>)
      scale Normal(<1,1,1>,<0.5,0.5,0.5>)

  every 5 frame:
    temps = [4500, 5000, 5700, 6000, 6300]
    with light:
      temperature Sequence(temps)

writers:
  BasicWriter:
    rgb = true
    semantic_segmentation = true
```

Come prima, procediamo a compilare lo script. Questa volta usiamo però il comando:

```
$ yarc compile hello_world.yrc -o hello_world.py -w
```

Così facendo andiamo ad attivare il controllo dei warning. YARC, infatti, stamperà a schermo un warning di tipo `MissingWriterParameter` dicendo che la cartella di output del `BasicWriter` non è stata definita.

I warning segnalano comportamenti da evitare ma non impediscono al compilatore di generare il codice eseguibile. Per esempio, YARC può scegliere autonomamente in quale cartella inserire i dati generati ma è consigliato che sia l'utente a definirlo. I vari tipi di warning e le loro cause sono specificati nella sezione 4.3.

Per risolvere il warning basta sostituire la sezione dei writer con la seguente:

```
writers:
  BasicWriter:
    output_dir = './dataset'
    rgb = true
    semantic_segmentation = true
```

3.4 Pieno controllo sulla tua scena

Creiamo ora uno scenario più complesso, dall'inizio alla fine, in modo da introdurre gli ultimi concetti fondamentali per la costruzione di una scena e la generazione di dati sintetici.

Stavolta andremo a comporre una scena usando asset e modelli preesistenti, caso d'uso che si renderà necessario il più delle volte. Vogliamo creare dati sintetici per un task di riconoscimento di automobili in ambiente urbano, abbiamo a disposizione le seguenti risorse:

- `env.usd`: una ricostruzione di un ambiente urbano, composta da pavimento stradale, segnali e altri oggetti che rendono lo scenario più realistico come pedoni e vegetazione
- `models/*`: una serie di modelli 3D di diversi tipi di automobili

Tutti le risorse si trovano all'interno della cartella al percorso `'home/yarc/car_detection'`. Introduciamo l'ultima sezione di uno script YARC, i setting, dove possiamo definire variabili globali e impostazioni per il render. In questo caso ci interessa definire il numero di scene e il punto di mount. Il punto di mount è un modo comodo per definire dove tutte le nostre risorse si trovano. La nostra sezione di setting è la seguente:

```
settings:
  num_scenes = 2500
  mount = 'home/yarc/car_detection/'
```

Nella sezione di `stage` andiamo a caricare la nostra ambientazione e a fare il fetching dei modelli di automobile da istanziare nella scena:

```
stage:
  open '*/env.usd'

  car_models = fetch 'usd' from '*/models/' recursive
```

Tutte le risorse possono essere referenziate usando il percorso relativo al punto di mount, usando `'*/'` come segnaposto del punto di mount stesso. Il fetching colleziona tutti i file con l'estensione indicata all'interno del percorso indicato, con `recursive` si richiede di effettuare la ricerca anche nelle sottocartelle.

Decidiamo di identificare anche i pedoni, per fare questo possiamo cercare oggetti sulla base del loro tipo e del loro path all'interno della scena:

```
  pedestrian = get 'Street/Pedestrian'
  edit pedestrian:
    semantics 'class:pedestrian'
```

Ogni frame devono essere scelte a caso 5 automobili e posizionate all'interno delle scena. Tutto ciò può essere facilmente ottenuto con l'istruzione `instantiate`:

```
  every frame:
    cars = instantiate 5 from car_models:
      translate to Uniform(<0,0,0>,<500,500,0)
      semantics 'class:car'
```

Creiamo un `BasicWriter` e settiamone i dati in output:

```
writers:
  BasicWriter:
    rgb = true
    bounding_box_2d_tigth = true
```

Lo script completo è il seguente (non dimentichiamoci della riga di intestazione):

```
scenario CarDetection: Replicator

settings:
  num_scenes = 2500
  mount = 'home/yarc/car_detection/'
stage:
  open '*/env.usd'
```

```
  car_models = fetch 'usd' from '*/models/'

  pedestrian = get 'Street/Pedestrian'
  edit pedestrian:
    semantics 'class:pedestrian'

  every frame:
    cars = instantiate 5 from car_models:
      translate to Uniform(<0,0,0>,<500,500,0)
      semantics 'class:car'

writers:
  BasicWriter:
    rgb = true
    bounding_box_2d_tigth = true
```

3.5 Uso avanzato

In quest'ultimo esempio andremo a vedere alcune delle funzionalità più avanzate di YARC.

Per prima cosa, creeremo tre tipi di oggetti diversi e li posizioneremo casualmente su di un piano. Per fare questo possiamo fare uso di due nuove istruzioni: `group` e `scatter2d`.

```
stage:
  cubes = create 5 Cube
  cones = create 5 Cone
  torus = create 5 Torus

  plane = create Plane

  shapes = group [cubes, cones, torus]:
    scatter2d on plane:
      check_for_collisions true
```

`group` permette di applicare istruzioni comuni a gruppi di oggetti, in modo da compattare il codice, `scatter2d` invece è una regola composta: queste di solito coinvolgono due o più modelli e possono ricevere dei parametri per definirne il comportamento.

Altre regole complesse includono la rotazione di un oggetto attorno ad un altro e tutto ciò che riguarda la fisica simulata. Vediamo ora queste ultime, introducendo inoltre la seconda versione del blocco `every`.

```
with plane:
  collider

every 3 seconds:
  edit shapes:
    translate z Normal(250,10)
    kinematics:
      velocity Uniform(-25,-10)
```

In questo esempio andiamo ad attivare un collider sul piano di appoggio (in modo tale che gli oggetti non vi possano passare attraverso). Come si può notare, un attributo composto può anche essere settato senza parametri. Gli altri oggetti, ogni 3 secondi, vengono traslati verso l'alto e poi fatti cadere verso il basso.

Per gli utenti più esperti, è data la possibilità di definire degli snippet in codice nativo. Questi possono essere ovunque, tranne all'interno di un blocco di `edit`, e le variabili dichiarate al loro interno possono essere riutilizzate all'esterno dello snippet. Il codice di uno snippet viene posto tra `'{{'` e `'}}'`, per esempio:

```
stage:
  {{ n_cube = 1 }}
  cube = create n_cube Cube
```

4 Reference

In quest'ultima sezione saranno presentati dettagliatamente:

- Il tool a riga di comando.
- La sintassi del linguaggio.
- Errori, warning e come risolverli.

4.1 CLI Tool

Il tool a riga di comando è lo strumento principale per la creazione di codice eseguibile a partire da script YARC. È sempre possibile avere accesso all'help digitando l'opzione `--help`.

Utilizzo:

```
$ yarc [OPZIONI] COMANDO [ARGOMENTI]...
```

Opzioni:

- `-v, --version`: Stampa la versione di YARC e termina.
- `--help`: Mostra il messaggio di help e termina.

Comandi:

- `compile`
- `translate-grammar`

4.1.1 yarc compile

È il comando principale del tool, permette di tradurre uno script YARC in codice eseguibile. Se dovessero essere presenti uno o più errori, li mostra a schermo e interrompe l'esecuzione.

Utilizzo:

```
$ yarc compile [OPZIONI] INPUT
```

Argomenti:

- `INPUT`: Percorso del file YARC da tradurre.

Opzioni:

- `-o, --output`: Percorso del file dove salvare il codice tradotto. Se non presente, stampa a schermo.
- `-w, --warnings`: Flag per indicare se mostrare i warning.
- `-l, --lib, --library`: Sovrascrive la libreria indicata nel file in input.
- `-n, --num-scenes`: Sovrascrive il numero di scene indicato nel file in input.
- `-m, --mount`: Sovrascrive il punto di mount indicato nel file in input.
- `--help`: Mostra il messaggio di help e termina.

4.1.2 yarc translate-grammar

Questo commando è invece di interesse per gli sviluppatori che fossero intenzionati a portare YARC su altre piattaforme. Per ridurre al minimo lo sforzo, questo comando di utility produce una versione della grammatica di YARC nel linguaggio specificato.

Usage:

```
$ yarc translate-grammar [OPZIONI] LANGUAGE OUTPUT_DIR
```

Argomenti:

- `LANGUAGE:{Java|Cpp|CSharp|Python2|Python3|JavaScript|Ruby}`: Linguaggio target per la grammatica.
- `OUTPUT_DIR`: Percorso della cartella di output in cui salvare la grammatica.

Opzioni:

- `--help`: Mostra il messaggio di help e termina.

4.2 Scene Description Language

In questa sezione andremo a descrivere il linguaggio di YARC. Questa sezione non contiene spiegazioni su come scrivere scene in YARC, spiega semplicemente le istruzioni e la loro sintassi.

La notazione che andremo a utilizzare è la seguente:

- `|` indica *alternativa*
- `*` indica *iterazione* (zero o più volte)
- `+` indica *iterazione* (una o più volte)

- `?` indica *opzionalità* (zero o una volta)
- `..` indica un *intervallo* di caratteri

4.2.1 Keywords

YARC ha una lista di keywords riservate, che non possono essere usate come identificatori e vanno scritte esattamente come sono di seguito riportati.

A			
and	at		

C			
Camera	Choice	collider	Combine
Cone	create	Cube	Cylinder

D			
Disk			

E			
edit	else	every	

F			
fetch	for	frame	from

G			
get	Group		
I			
in	instantiate	is	
K			
kinematics			
L			
len	Light	limit	LogUniform
look_at			
M			
match	Material	material	move_to_camera
N			
none	Normal	not	
O			
on	open	or	

P			
physics_material	pivot	Plane	

R			
recursive	rigid_body	rotate	rotate_around

S			
scale	scatter_2d	scatter_3d	scenario
second	semantics	Sequence	settings
size	Sphere	stage	step
Stereo			

T			
Timeline	to	Torus	translate
true			

U	
Uniform	up_axis

V	
visible	

W	
writers	

X	
x	X

Y	
y	Y

Z	
z	Z

4.2.2 Letterali

I letterali sono notazioni specifiche per tipi nativi. YARC supporta tre tipi di letterali:

- Numeri interi
- Numeri con la virgola
- Stringhe

4.2.2.1 Numeri interi

I numeri interi sono descritti dalle seguenti regole lessicali.

```
INTEGER:
  NON_ZERO_DIGIT DIGIT*
  | '0'+                       // Decimal integer
  | '0' ('b' | 'B') BIN_DIGIT+ // Binary integer
  | '0' ('o' | 'O') OCT_DIGIT+ // Octal integer
  | '0' ('x' | 'X') HEX_DIGIT+ // Hexadecimal integer
;
NON_ZERO_DIGIT : '1' .. '9';
DIGIT          : '0' .. '9';
BIN_DIGIT      : '0' | '1';
OCT_DIGIT      : '0' .. '7';
HEX_DIGIT      : DIGIT | 'a' .. 'f' | 'A' .. 'F';
```

Non c'è limite alla lunghezza dei letterali. Non sono permessi zeri iniziali in un numero che sia diverso da 0. Gli interi si possono rappresentare in decimale, binario (prefisso `0b`), ottale (prefisso `0o`) e in esadecimale (prefisso `0x`).

4.2.2.2 Numeri con la virgola

I numeri con la virgola sono descritti dalle seguenti regole lessicali:

```
FLOAT_NUMBER   : POINT_FLOAT | EXPONENT_FLOAT;
POINT_FLOAT    : INT_PART? FRACTION | INT_PART '.';
EXPONENT_FLOAT : ( INT_PART | POINT_FLOAT) EXPONENT;
INT_PART       : DIGIT+;
FRACTION       : '.' DIGIT+;
EXPONENT       : ('e' | 'E') ('+' | '-')? DIGIT+;
```

I numeri con la virgola si possono rappresentare in notazione puntata o con la notazione esponenziale. Sia la parte intera sia l'esponente sono sempre interpretati in base 10.

4.2.2.3 Stringhe

Le stringhe sono descritte dalle seguenti regole lessicali:

```
STRING: STRING_PREFIX? SHORT_STRING;
STRING_PREFIX:
  ('u' | 'U')
  | (('f' | 'F')? ('r' | 'R'))
  | (('r' | 'R')? ('f' | 'F'))
;
SHORT_STRING:
  "'" SHORT_STRING_ITEM* "'" | '"' SHORT_STRING_ITEM* '"'
;
SHORT_STRING_ITEM: SHORT_STRING_CHAR | STRING_ESCAPE_SEQ;
SHORT_STRING_CHAR: <any character except '\' or newline or the quote>;
STRING_ESCAPE_SEQ: '\' <any character>;
```

Le stringhe possono quindi essere racchiuse tra singoli apici (') o doppi apici ("). Il carattere `'\'` è usato per fare escaping di caratteri che altrimenti avrebbero significati speciali, come l'a capo, il backslash stesso o gli apici.

Le stringhe possono essere prefisse con la lettera `'r'`, queste sono dette *raw strings* e considerano i backslash come letterali. Se il prefisso è la lettera `'u'` si intende una stringa codificata Unicode mentre se il prefisso è la lettera `'f'` si possono inserire dei placeholder che a runtime saranno sostituiti dal valore assegnato al placeholder. Questa funzionalità è derivata direttamente da Python, per cui si rimanda alla documentazione ufficiale[1]

4.2.3 Identificatori

In YARC si possono definire identificatori da usare nello script. Un identificatore è formato da lettere maiuscole e minuscole, le cifre da 0 a 9 e il simbolo di underscore `'_'`. Il primo carattere dev'essere un carattere alfabetico o l'underscore. Non c'è un limite alla lunghezza degli identificatori. Non possono coincidere con le keywords.

4.2.4 Commenti

Un commento è una stringa di testo in uno script che ne spiega alcune porzioni e lo rende più facile da leggere. Sono quindi utili solo a un eventuale lettore e sono ignorati dal compilatore. Un commento inizia con il carattere hash (#) e termina alla fine della riga, non esistono quindi i commenti multilinea.

4.2.5 Struttura di uno script

Uno scenario YARC ha la seguente struttura:

```
scenario:
  declaration
  code_snippet*
  settings?
  stage
  writers?
  code_snippet*
;
```

[1] https://docs.python.org/3/reference/lexical_analysis.html#formatted-string-literals

La `declaration` è obbligatoria e serve a specificare il nome dello scenario. È formata dalla keyword `'scenario'` seguita dal nome dello scenario. Si può opzionalmente specificare la libreria target, digitando il simbolo di `':'` e specificandone il nome, nel seguente modo:

```
scenario name(: target_library)?
```

Un `code_snippet` è invece una porzione in codice nativo racchiuso tra doppie parentesi graffe (`{{...}}`). Se lo snippet dovesse essere multilinea, il blocco più esterno deve cominciare all'inizio della linea. Gli snippet vengono riportati nel file di output così come sono, per cui è compito dell'utente assicurarsi che questi siano quanto più corretti possibilr.

Dopo la `declaration`, vi sono le tre sezioni principali di un file YARC: `settings`, `stage` e `writers`. Solo la seconda è obbligatoria. Ogni sezione è introdotta dal nome della sezione stessa seguito dal simbolo di `':'`, tutte le istruzioni all'interno di una sezione sono in un blocco indentato.

Uno dei concetti fondamentali di YARC è la strutturazione del codice attraverso blocchi indentati. Ogni riga di un blocco di istruzioni annidate dev'essere allo stesso livello (cioè, dev'esserci lo stesso numero di spazi prima del primo carattere di quella riga). Per facilita di lettura si consiglia di usare due spazi come livello di indentazione.

Nella sezione di settings si possono definire le impostazioni del generatore di dati e le variabili globali. Per dichiarare un setting basta effettuare un assegnamento. La sezione di setting è quindi composta nel seguente modo:

```
settings:
  setting_name = value
  ...
```

dove *`setting_name`* è un identificatore e *`value`* il risultato di un'espressione. YARC definisce una serie di setting di default che è possibile sovrascrivere. Di seguito sono elencati i setting preimpostati, il valore di default tra parentesi e il significato:

- `mount` (`"."`), indica il percorso comune a tutte le risorse, serve a facilitare la scrittura degli script. Per indicare un percorso relativo al mount basta farlo iniziare con `'*/'`.

- `seed`, indica il seed da settare per tutte le operazioni di estrazioni di numeri. Se non specificato, gli viene assegnato un valore casuale durante la compilazione.
- `num_scenes` (`1`), indica il numero di scene da generare. Se tutti i trigger sono di tipo `frame`, coincide con il numero di frame, altrimenti è il numero di sequenze di frame da generare.
- `fps` (`24`), indica il numero di frame in ogni secondo.
- `stage_meters_per_unit` (`1`), indica a quanti metri corrisponde un'unità di spazio nell'ambiente simulato.
- `stage_up_axis` (`"y"`), indica qual è l'asse verticale nell'ambiente simulato.
- `resolution` (`[512, 512]`), indica la dimensione di output dei dati generati.

La sezione `stage` è invece composta da statement. Questi statement possono essere l'`open statement`, `expression statement`, `edit statement` o `behavior statement`. È inoltre possibile aggiungere degli snippet di codice. Ci può essere un solo `open statement` e dev'essere il primo statement della sezione di stage. L'`open statement` ha la seguente struttura:

```
open path_to_file
```

e si occupa di importare un intero file e di usarlo come ambientazione.

Le `expression statement` sono gli assegnamenti di un valore a un identificativo, in modo simile ai setting. Il valore può essere il risultato di un'espressione, una `fetch expression` o una `model expression`. Una `fetch expression` si occupa di collezionare tutti i file di una data estensione in una data cartella, è inoltre possibile specificare se filtrare (`match`) i file con un pattern, se limitare (`limit`) il numero di risultati e se cercare ricorsivamente (flag `recursive`) in tutte le sottocartelle:

```
fetch extension from path (match pattern_to_match)?
  (limit max_results)? recursive?
```

Le `model expression` riguardano rispettivamente la creazione, l'istanziazione, il raggruppamento o la ricerca di oggetti all'interno della scena. Si dovrebbe preferire l'istanziazione quando gli oggetti vengono creati dinamicamente e in quantità elevate, per indicare all'engine di gestirli nel modo più corretto. La sintassi è la seguente:

```
create quantity? (PRIMITIVE | from path_to_model)
```

```
instantiate quantity? from path_to_model

group [model, ...]

get (object_type at)? path_in_scene
```

Le primitive supportate nella `create` sono i seguenti:

- Plane
- Cube
- Cone
- Torus
- Sphere
- Cylinder
- Disk
- Camera
- Stereo Camera
- Light
- Material

I tipi di oggetti supportati nella `get` sono i seguenti:

- Camera
- Light
- Material

Ogni libreria supportata può estendere la lista di tipi supportata, per cui si rimanda alla documentazione specifica per ogni libreria target.

Tutti e quattro i tipi di statement possono specificare attributi o `behavior statement`, facendo seguire all'espressione il simbolo di `':'` e un blocco indentato:

```
model_expression:
  attribute | behavior_statement
  ...
```

In modo molto simile, un `edit statement` indica quali attributi o dinamica settare di un oggetto. Si può anche fare riferimento alla timeline per modificarne istanti di inizio e fine o portarla a un frame specifico (utile quando si trattano oggetti animati):

```
edit (object | Timeline):
  attribute | behavior_statement
  ...
```

I `behavior statement` indicano invece come modificare gli attributi di uno o più oggetti rispetto a un trigger (`frame` o `second`), ne esistono due versioni: la versione esterna e la versione interna a una `model expression`. Nella versione esterna si possono includere `model expression`, nella versione interna invece non si possono

includere altre `model expression` e tutti gli attributi fanno riferimento automaticamente all'oggetto definito dalla `model expression` stessa. La sintassi è la seguente:

```
every interval? (frame | second):
  attribute | model_expression
  ...
```

Esistono due versioni di attributi, gli attributi semplici e gli attributi composti. Gli attributi semplici sono coppie chiave-valore. Tra questi esiste un sottoinsieme di attributi core in quanto sono, normalmente, i più usati quando si fa Synthetic Data Generation. Se l'attributo non esiste per quell'oggetto, bisogna definirne anche il tipo (anche in questo caso, a seconda dell'engine). La definizione di un attributo è:

```
attribute(: type)? value
```

Di norma qualsiasi attributo è accettato ma solo alcuni hanno un effetto sui modelli, altri attributi invece modificano il comportamento della statement (ad esempio, l'attributo `exclude` per `get`). Per sapere quali sono questi attributi, fare riferimento alla documentazione specifica di YARC per la libreria scelta. Al contrario gli attributi core sono gli stessi per tutte le librerie e sono i seguenti:

- `translate` *`axis`*`? to`, trasla l'oggetto nella posizione specificata. Se specificato, muove solo lungo l'asse scelto.
- `rotate` *`axis`*`?`, ruota gli assi dell'oggetto degli angoli specificati. Se specificato, ruota solo l'asse scelto.
- `scale`, riscala gli assi dell'oggetto del fattore di scala scelto.
- `look_at`, orienta l'oggetto verso un punto o un altro oggetto.
- `up_axis`, modifica l'asse verticale dell'oggetto, utile per modificare il comportamento di `look_at`.
- `size`, riscala l'oggetto in modo da avere la dimensione specificata.
- `pivot`, modifica il punto di applicazione delle varie trasformazioni.
- `semantics`, etichetta l'oggetto con una serie di classi, utile per segmentazione e bounding box.
- `visible`, modifica la visibilità dell'oggetto.
- `material`, modifica il materiale dell'oggetto.

Gli attributi composti sono invece attributi che interessano l'interazione tra più oggetti oppure la simulazione fisica della scena. Sono considerati *composti* in quanto possono accettare a loro volta un blocco indentato di attributi che ne modifica il comportamento (con la solita sintassi). Gli attributi composti sono i seguenti:

- `scatter_2d`, posiziona uno o più oggetti a caso sulla superficie di un altro oggetto. La sintassi è:

```
scatter_2d on surface:
  attribute
  ...
```

- `scatter_3d`, posiziona uno o più oggetti a caso in modo che siano all'interno del volume di un altro oggetto. La sintassi è:

```
scatter_3d on volume:
  attribute
  ...
```

- `rotate_around`, posiziona e orienta un'oggetto in modo che segue una traiettoria circolare con al centro l'oggetto o un punto specificato. La sintassi è:

```
rotate_around center:
  attribute
  ...
```

- `move_to_camera`, posiziona l'oggetto rispetto al campo visivo della camera specificata. La sintassi è:

```
move_to_camera camera:
  attribute
  ...
```

- `collider`, crea un collisore attorno all'oggetto in modo che altri oggetti non gli passino attraverso. La sintassi è:

```
collider:
  attribute
  ...
```

- `rigid_body`, setta tutti i parametri di corpo rigido di un oggetto (es. massa, densità, inerzia, …). La sintassi è:

```
rigid_body:
  attribute
  ...
```

- `kinematics`, setta la cinematica dell'oggetto (es. velocità). La sintassi è:

```
kinematics:
  attribute
  ...
```

- `physics_material`, crea un materiale con proprietà fisiche (es. attrito) per l'oggetto. La sintassi è:

```
physics_material:
  attribute
  ...
```

Quando si settano più attributi è possibile che questi vadano in conflitto gli uni con gli altri visto che agiscono sugli stessi parametri dell'oggetto. Quando si crea un conflitto, solo l'ultimo attributo avrà effetto (gli altri saranno sovrascritti). Gli attributi in conflitto sono:

- `translate`, `scatter_2d`, `scatter_3d`, `move_to_camera`, `rotate_around` (conflitto sulla posizione).
- `rotate`, `look_at`, `rotate_around` (conflitto sull'orientamento).
- `scale`, `size` (conflitto sulla dimensione).

Infine, nella sezione `writers` è possibile effettuare assegnamenti e istanziare writer. Gli assegnamenti sono simili alle `expression statement` dello `stage`, con la sola eccezione che non si possono definire `model expression`. La sezione è quindi così formata:

```
writers:
  writer | expression_statement
  ...
```

Un writer è istanziato semplicemente con il suo nome, seguito da un blocco indentato di assegnamenti che ne impostano i parametri di generazione, come nel seguente esempio:

```
writer_name:
  parameter = value
  ...
```

in modo molto simile alla sezione `settings`. È inoltre possibile definire degli snippet di codice. I writer supportati e i loro parametri dipendono dalla specifica libreria, per cui si chiede di fare riferimento alla documentazione specifica per la libreria scelta.

4.2.5.1 Espressioni

L'ultimo parte del linguaggio da definire sono le espressioni e gli assegnamenti. Iniziamo dagli operatori:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=
```

Oltre a questi, si vanno ad aggiungere: `and`, `or`, `not`, `is`, `in`, `is not`, `not in`.

L'assegnamento semplice è il simbolo di `'='` ma esistono anche le versioni *augmented* che prima di assegnare eseguono l'operazione relativa:

```
+=      -=      *=      **=     /=      //=     %=
<<=     >>=     &=      |=      ^=
```

Un'espressione è costituita da atomi che possono essere composti attraverso gli operatori, come nelle classiche espressioni matematiche, facendo anche uso delle parentesi per modificare la precedenza degli operatori.

Per descrivere gli atomi, cominciamo dai tipi supportati da YARC:

- Stringhe
- Numeri interi
- Numeri con la virgola
- Booleani (`true`, `false`)
- Il tipo nullo (`none`)
- Distribuzioni di probabilità
- Collezioni:
    - Vettori
    - Liste ordinate
    - Range
    - Insiemi (senza ripetizioni)
    - Dizionari (coppie chiave-valore)

Per dichiarare una distribuzione di probabilità basta definirne il tipo e passare i parametri necessari tra parentesi. La sintassi è quindi la seguente:

```
distribution(parameter,...)
```

Le distribuzioni supportate sono:

- `Uniform`, accetta un limite inferiore e un limite superiore. Estrae secondo una uniforme.
- `LogUniform`, accetta un limite inferiore e un limite superiore. Estrae secondo una log-uniforme.
- `Normal`, accetta una media e una varianza. Estrae secondo una normale.
- `Choice`, accetta una lista di valori. Estrae un valore a caso dalla lista.
- `Sequence`, accetta una lista di valori. Estrae i valori in sequenza.
- `Combine`, accetta una lista di distribuzioni. Estrae un valore per ognuna.

Se alle prime tre distribuzioni vengono passate delle collezioni di valori invece che parametri singoli, verrà estratto un valore per ogni elemento.

I vettori sono tuple di 3 elementi racchiusi tra parentesi angolari (`<value,value,value>`), si usano principalmente quando si tratta di operazioni che hanno a che fare con le coordinate o gli assi degli oggetti.

Le liste ordinate sono racchiuse tra parentesi graffe e possono essere costruite in due modi:

- Elencando gli elementi, `[value, value, ...]`
- Tramite *list comprehension*, con la seguente sintassi:

```
[expression for item in collection if condition]
```

I range sono racchiusi tra parentesi quadre e sono formate da un valore di inizio, un valore di fine e opzionalmente un passo, separati da `':'`, nel seguente modo:

```
[start:stop(:step)?]
```

Gli insiemi sono collezioni non ordinate e senza duplicati, si possono costruire come le liste, con l'unica differenza che devono essere racchiusi tra parentesi graffe.

I dizionari sono anch'essi racchiusi tra graffe ma rappresentano coppie chiave-valore, possono essere costruiti nei seguenti modi:

- Elencando le coppie, `{key: value, key: value, ...}`
- Tramite *dict comprehension*, con la seguente sintassi:

```
{key_expr: expr for item in collection if condition}
```

Si possono indicizzare le liste ordinate, per selezionare solo uno oppure un range di elementi. La sintassi è molto simile a quella dei range, con la sola differenza che tutti le parti sono opzionali. L'assenza dello start implica di iniziare dal primo elemento, l'assenza dello stop implica di arrivare fino all'ultimo elemento. Si può indicizzare una lista nel seguente modo:

```
list[start:stop:step]
```

Per indicizzare un dizionario allo scopo di recuperare elemento, basta usare la sua chiave, in notazione puntata:

```
dict.key
```

Infine, si può calcolare la lunghezza di una collezione richiamando il metodo len(...) su una collezione, nel seguente modo:

```
len(collection)
```

Tutti i tipi supportati sono atomi, comprese le operazioni di indicizzazione e calcolo della lunghezza. Sono atomi anche gli identificatori e gli identificatori a variabili globali definite nei `settings` (per richiamarle basta prefissare il simbolo di `'$'` al nome del setting).

Le espressioni sono quindi composizioni di un qualsiasi numero di atomi attraverso gli operatori. Esiste infine l'ultimo modo di creare un'espressione, attraverso le `conditional expressions`. Queste, data una condizione, assumono il primo valore specificato se la condizione è vera, altrimenti assumono il secondo valore. La sintassi è:

```
expression if condition else expression
```

4.3 Errori e Warning

Quando si compila uno script può capitare che vengano generati messaggi di errore o di warning. Questi messaggi sono strumenti utili per identificare e risolvere eventuali problemi nel codice YARC. Di seguito andremo a descrivere i tipi di errori e le possibili cause.

È importante capire innanzitutto qual è la differenza tra errore e warning:

- Un errore è una situazione irreparabile che non permette al compilatore di tradurre lo script.
- Un warning indica invece una situazione che non è problematica ma che richiede comunque attenzione.

YARC cerca di essere il più esplicito possibile: ogni errore o warning ha una breve descrizione che indica anche la porzione di testo che lo ha generato, favorendo e semplificando l'operazione di debugging dello script.

Gli errori che possono essere generati durante il processo di compilazione sono i seguenti:

- **SyntaxError**: generato quando si commette un errore sintattico nello script, per esempio non rispettando la struttura dei blocchi di edit o quando si utilizza una parola chiave dove non dovrebbe essere usata. Per esempio:

```
settings:
  mount = 'path/to/mount' num_scenes = 200
```

```
Cube = rep create Cube
```

- **IndentationError**: viene generato ogni volta che si commette un errore nell'indentazione dei blocchi. Può essere causato da un cambio del livello di indentazione quando non richiesto o dalla mancanza di indentazione di un blocco innestato. Per esempio:

```
edit shape:
scale 2
```

```
writers:
  BasicWriter:
    rgb = true
      output_dir = '*/out'
```

- **LibraryError**: viene generato quando viene richiesta una libreria non riconosciuta da YARC. Per esempio:

```
scenario HelloWorld: NonExistentLib
```

- **SnippetError**: per essere sicuri della correttezza del codice generato, anche gli snippet di codice sono sottoposti a un controllo sintattico. Se lo snippet contiene un errore, verrà generato un errore di questo tipo. Per esempio:

```
{{ print('This is wrong }}
```

- **SettingError**: viene generato quando si fa riferimento a un setting non dichiarato. Per esempio:

```
settings:
  my_setting = 12.34
stage:
  set = $non_existent_setting
```

- **WriterError**: viene generato quando il writer selezionato non è tra quelli supportati della libreria scelta. Per esempio:

```
writers:
  NonExistentWriter:
    rgb = true
```

- **NameError**: viene generato quando una variabile viene utilizza prima di essere dichiarata o quando il nome di una variabile coincide con una keyword del codice nativo. Per esempio:

```
stage:
  create num_cubes Cube
```

```
False = false
```

I warning che possono essere generati durante la compilazione sono:

- **UnusedVariable**: viene generato quando una variabile viene dichiarata ma non viene mai utilizzata.
- **UnknownParameter**: viene generato quando nel blocco di `edit` ci sono attributi che il compilatore non sa come gestire e quindi ignora. Se possibile, viene suggerito l'attributo con il nome più simile.
- **UnsupportedTarget**: viene generato quando si cerca nello stage una tipologia di oggetto non riconosciuta dall'engine. Anche in questo caso viene suggerita, quando possibile, la tipologia con il nome più simile.
- **DuplicatedSetting**: generato quando una variabile globale viene settata due o più volte nello stesso script.
- **OverwrittenAttributes**: generato quando in un blocco di `edit` vengono modificati più volte gli stessi parametri dell'oggetto. Per esempio:

```
stage:
  create Cube:
    rotate <0,0,90>
    look_at <-10,-10,-10>
```

In questo caso entrambi gli attributi andrebbero a modificare l'orientamento del cubo ma solo il secondo avrà effetto.

- **MissingWriterParameter**: viene generato quando alcuni attributi dei writer non sono stati specificati ma possono essere settati di default (es. il percorso in cui salvare i dati generati).