Documentazione Tecnica

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

1.1 Obiettivi del progetto

Lo scopo del progetto è realizzare un compilatore per la traduzione di script, definiti attraverso uno Scene Description Language, in file eseguibili all'interno di un engine di modellazione 3D e simulazione per la generazione di dataset sintetici (Synthetic Data Generation). Sono quindi stati realizzati un linguaggio ad hoc per la definizione di ambienti 3D e della loro *dinamica* (es. in che coordinate posizionare un oggetto ad ogni frame della generazione), chiamato YARC, e un tool da riga di comando per la traduzione automatica dello script con controllo degli errori. Il file prodotto può essere eseguito direttamente nell'engine scelto.

1.2 Synthetic Data Generation

Con Synthetic Data Generation (SDG) si intende una serie di tecniche, algoritmi e strumenti per generare informazioni automaticamente annotate, come immagini o testo, allo scopo di allenare modelli di Machine Learning. Questi, infatti, necessitano di una grande quantità di dati per poter raggiungere livelli di precisione e performance elevati. Spesso i dati reali sono difficili da raccogliere sul campo e ancora più complesso e costoso è il processo di etichettatura dei dati. Si pensi per esempio a un modello di visione per attività di bin picking, anche di una sola tipologia di oggetto: sono necessarie molte acquisizioni diverse tra loro e, una volta terminata l'acquisizione, ogni oggetto presente all'interno di ciascuna immagine dev'essere segmentato. Tecniche di SDG permettono di risparmiare tempo, ridurre i costi e aumentare a piacere la variabilità dei dati, in modo che il modello impari a generalizzare meglio in produzione.

Seppur uno strumento molto potente, i dati sintetici sono difficili da ottenere in quanto spesso mancano gli strumenti adatti per poter generare e annotare i dati. Uno sforzo da questo punto di vista è stato svolto da NVIDIA, che ha recentemente rilasciato Omniverse, un framework per la modellazione 3D, mettendo a disposizione un'ampia varietà di strumenti a supporto degli sviluppatori, tra i quali figura Replicator. Replicator è una libreria in Python pensata appositamente per la definizione di ambienti da cui generare dati e groundtruths correttamente etichettate (es. immagini segmentate,

bounding box in 2D e 3D, depth map, …) tramite semplici script. NVIDIA ha rilasciato anche Replicator Composer, una utility che genera i dati sintetici a partire da un file YAML strutturato secondo quanto descritto nella documentazione. Il problema principale di Composer è che basandosi su YAML e non su una grammatica ad hoc la creazioni di scenari complessi risulta molto difficile e poco conveniente.

Si può immaginare che altri importanti attori, come già Blender e Unity, si muoveranno nella stessa direzione e renderanno disponibile strumenti per la generazione di dati sintetici.

1.3 YARC

Uno dei possibili sviluppi negativi derivati dall'interesse nei synthetic data è la proliferazione di strumenti proprietari e librerie non intercambiabili. Ciò implica che la scelta di uno strumento è vincolante e diventa difficile imparare ad usare uno strumento diverso per sfruttarne al meglio le potenzialità quando necessario. Un altro svantaggio deriva dal fatto che l'ambientazione spesso necessita di molti modelli 3D per risultare verosimile, la procedura di generazione va quindi divisa in due parti:

- la creazione degli asset e la composizione di scenari complessi, effettuata da un 3D designer;
- lo sviluppo dello script per la generazione dei dati, effettuata da un developer che conosce la libreria.

Questo processo può risultare complesso e lento, sarebbe sicuramente più efficiente avere una sola figura che si occupa dell'intero processo.

YARC (Yarc Ain't Replicator Composer) vuole fare fronte proprio a questi due problemi: è un linguaggio agnostico, indipendente cioè dalle librerie attualmente disponibili, e non si presenta come un tipico linguaggio di programmazione, segue invece un approccio dichiarativo per quanto riguarda la composizione della scena, in modo da poter essere agevolmente usato da chiunque, senza dover necessariamente essere uno sviluppatore esperto, risultando di facile approccio e molto intuitivo.

2 Lessico e Sintassi

La grammatica di YARC è stata sviluppata in ANTLR 3. ANTLR (ANother Tool for Language Recognition) è un generatore di lexer e parser che permette di creare analizzatori sintattici per diversi linguaggi di programmazione. ANTLR offre molti vantaggi, tra i quali facilità di sviluppo, flessibilità e potenza nell'analisi del linguaggio.

Il linguaggio target scelto è Python 3, in modo da essere immediatamente compatibile e integrabile con le librerie e gli engine già presenti. Questa decisione ha portato innanzitutto a dover adattare le librerie di ANTLR 3 stesso e StringTemplate 3 (una libreria per la generazione di testo a partire da template) in quanto entrambe sviluppate prima del rilascio di Python 3 e compatibili solamente con la versione 2. La maggior parte della traduzione è stata eseguita automaticamente da `2to3`, una utility messa a disposizione da Python stesso. Le versioni aggiornate si possono trovate nelle rispettive repository[1].

YARC è fortemente ispirato a Python, dal momento che vuole essere snello e di facile comprensione. Proprio per questo motivo, parte del lessico e della sintassi sono ereditati direttamente dall'implementazione ufficiale della grammatica di Python 3 in ANTLR[2].

Per semplicità di sviluppo, lexer e parser sono stati separati in due file diversi. Per ridurre al minimo il codice nativo presente all'interno dei file di specifica della grammatica e facilitare un possibile cambiamento del linguaggio target, entrambi gli analizzatori ereditano da classi base che implementano i metodi necessari al corretto funzionamento.

2.1 Lessico

Come già anticipato, parte del lessico di YARC deriva da Python. Per semplicità si è deciso di utilizzare solo un sottoinsieme dei token (es. non sono stati mantenuti i token per la definizione di classi) e in alcuni casi il token riconosciuto non è capitalizzato (es., al posto di `True` e `False` vengono riconosciuti `true` e `false`).

[1] ANTLR 3 – Python 3: https://github.com/fcarne/antlr3-python3-runtime/
StringTemplate 3 – Python 3: https://github.com/fcarne/stringtemplate3-python3-runtime
[2] https://github.com/antlr/grammars-v4/tree/master/python/python3

In quanto segue sono riportati tutti i token riconosciuti dal lexer, per brevità non sono riportati i fragment che compongono i token. La notazione di ANTLR 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

Da Python vengono ereditati i seguenti token:

- Un sottoinsieme delle keywords.

```
IF         : 'if';
IN         : 'in';
IS         : 'is';
LEN        : 'len';
NONE       : 'none';
NOT        : 'not';
OR         : 'or';
TRUE       : 'true';
UNDERSCORE : '_';
```

- Gli operatori matematici e booleani.

```
BIT_OR  : '|';
XOR     : '^';
BIT_AND : '&';
BIT_NOT : '~';
LSHIFT  : '<<';
RSHIFT  : '>>';
PLUS    : '+';
MINUS   : '-';
MUL     : '*';
DIV     : '/';
MOD     : '%';
IDIV    : '//';
POWER   : '**';
```

- L’operatore di assegnamento e gli *augmented assignment operators.*

```
ASSIGN : '=';
AUG_ASSIGN:
  '+='
  | '-='
  | '*='
  | '/='
  | '%='
  | '&='
  | '|='
  | '^='
  | '<<='
  | '>>='
  | '**='
  | '//='
;
```

- Gli operatori di confronto.

```
LT     : '<';
GT     : '>';
EQUALS : '==';
GT_EQ  : '>=';
LT_EQ  : '<=';
NOT_EQ : '!=';
```

- Letterali, come stringhe, numeri interi e numeri float, comprese le varianti. I delimitatori di stringa sono sia i singoli apici (`'...'`) sia i doppi apici (`"..."`).

```
STRING: (
  ('u' | 'U')
  | ( ('f' | 'F')? ('r' | 'R'))
  | ( ('r' | 'R')? ('f' | 'F'))
  )? SHORT_STRING
;
INTEGER:
  NON_ZERO_DIGIT DIGIT*
  | '0'+                        // Decimal integer
  | '0' ('o' | 'O') OCT_DIGIT+ // Octal integer
  | '0' ('x' | 'X') HEX_DIGIT+ // Hexadecimal integer
  | '0' ('b' | 'B') BIN_DIGIT+ // Binary integer
;
FLOAT_NUMBER:
  POINT_FLOAT
  | EXPONENT_FLOAT;
```

I token appena definiti saranno usati per costruire le espressioni del linguaggio.

Vengono inoltre riconosciuti i classici token di punteggiatura e parentesi:

```
DOT      : '.';
COMMA    : ',';
COLON    : ':';

LPAREN : '(';
RPAREN : ')';
LBRACK : '[';
RBRACK : ']';
LBRACE : '{';
RBRACE : '}';
```

In YARC viene data la possibilità di definire variabili. Gli identificatori si compongono di un primo carattere a scelta tra `'_'` e `[a-zA-Z]` e opzionalmente di una combinazione di lettere, numeri e `'_'`. Si può inoltre fare riferimento ad alcune variabili settate globalmente giustapponendo il simbolo `'$'` al nome della variabile:

```
ID         : ID_START ID_CONTINUE*;
SETTING_ID : '$' ID;
```

Un file YARC è organizzato nel seguente modo: un header iniziale e tre macro-sezioni, ognuna introdotta da una parola chiave:

```
SCENARIO : 'scenario';
SETTINGS : 'settings';
STAGE    : 'stage';
WRITERS  : 'writers';
```

Per la definizione degli oggetti che compongono l'ambientazione esistono degli oggetti primitivi, presenti in tutti gli engine di modellazione 3D:

```
SHAPE :
  'Plane'
  | 'Cube'
  | 'Cone'
  | 'Torus'
  | 'Sphere'
  | 'Cylinder'
  | 'Disk'
;
CAMERA   : 'Camera';
LIGHT    : 'Light';
STEREO   : 'Stereo';
MATERIAL : 'Material';
TIMELINE : 'Timeline';
```

Per la specifica della dinamica della scena vengono resi invece disponibili dei token per la definizione di distribuzioni di probabilità:

```
DISTRIBUTION:
  'Uniform'
  | 'Normal'
  | 'Choice'
  | 'Sequence'
  | 'LogUniform'
;
COMBINE : 'Combine';
```

Sono stati inoltre definiti token per l'esecuzione di direttive per la specifica dello scenario:

- Apertura di ambientazioni esistenti e creazione di oggetti all'interno della scena.

```
OPEN        : 'open';
CREATE      : 'create';
GROUP       : 'group';
INSTANTIATE : 'instantiate';
GET         : 'get';
EDIT        : 'edit';
```

- Ricerca di risorse come materiali, texture, modelli 3D da importare.

```
FETCH     : 'fetch';
MATCH     : 'match';
LIMIT     : 'limit';
RECURSIVE : 'recursive';
```

- Attributi principali degli oggetti che compongono la scena (es. posizione, rotazione, scala, materiale).

```
TRANSLATE : 'translate';
ROTATE    : 'rotate';
SCALE     : 'scale';
SEMANTICS : 'semantics';
VISIBLE   : 'visible';
SIZE      : 'size';
LOOK_AT   : 'look_at';
UP_AXIS   : 'up_axis';
PIVOT     : 'pivot';
MATERIAL_ : 'material';
AXIS      : 'x' | 'y' | 'z' | 'X' | 'Y' | 'Z';
```

- Comportamenti complessi degli oggetti, come la definizione della fisica per ogni oggetto.

```
SCATTER     : 'scatter_' ('2d' | '3d');
ROT_AROUND  : 'rotate_around';
MOVE_TO_CAM : 'move_to_camera';
PHYSICS:
  'collider'
  | 'kinematics'
  | 'rigid_body'
  | 'physics_material'
;
```

- Definizione di trigger per dare comportamenti dinamici agli oggetti della scena.

```
EVERY  : 'every';
FRAMES : 'frame' 's'?;
TIME   : 'sec' ('ond' 's'?)?;
```

Per rendere YARC più simile al linguaggio naturale sono stati introdotti token ausiliari al solo scopo di chiarire la semantica delle varie direttive:

```
TO : 'to'; // translate to
ON : 'on'; // scatter on
AT : 'at'; // get ... at ...
```

Inoltre, viene data la possibilità di definire snippet in codice nativo, in modo che utenti esperti possano sopperire a eventuali funzionalità mancanti rispetto a quelle fornite dalla libreria target che si è deciso di usare. Si definisce snippet tutto ciò che è incluso tra doppie parentesi graffe `{{ ... }}`.

```
SNIPPET : NESTED_CODE;

fragment NESTED_CODE:
  LBRACE LBRACE
    ( options {k=2; greedy=false;}: NESTED_CODE | . )*
  RBRACE RBRACE
;
```

Come anche in Python si è deciso di ignorare la presenza di spazi e commenti ma non quella degli a capo, in quanto questi serviranno a definire dove inizia un *blocco indentato* di istruzioni (si veda la Sezione 2.1.1).

```
NEWLINE : ( ( '\r'? '\n' | '\r' | '\f') SPACES?);
SKIP_   : ( SPACES | COMMENT ) -> {self.skip()};
```

Infine, per quanto riguarda gli errori lessicali (token non riconosciuti), si è deciso di delegarli alla gestione sintattica:

```
UNKNOWN : .;
```

2.1.1 Indentazione

Come già detto, YARC eredita molto da Python. Tra le altre cose, eredita anche la definizione di blocchi di istruzioni tramite indentazione: due istruzioni fanno parte dello stesso blocco di codice se hanno lo stesso livello di indentazione (cioè, numero di spazi prima del primo carattere). Per poter gestire correttamente i blocchi indentati sono stati introdotti due ulteriori token:

```
tokens {
  INDENT;
  DEDENT;
}
```

Questi non saranno mai presenti nel file in input ma sono prodotti automaticamente dal lexer. Il lexer verifica, a fronte di un a capo se il livello di indentazione è cambiato: se così fosse genera il token adatto secondo il tipo di cambiamento (se è positivo, crea `INDENT` altrimenti `DEDENT`), in caso contrario ignora l'a capo, in modo tale da non considerare gli a capo che non portano nuova informazione e semplificare la sintassi quando possibile. Un a capo viene ignorato anche quando si è all'interno di un blocco tra parentesi, i seguenti input sono quindi considerati identici, con la sola differenza che il secondo può risultare più leggibile.

```
int_list = [0,1,2,3,4,5]
```

```
int_list = [
  0,
  1,
  2,
  3,
  4,
  5
]
```

2.2 Sintassi

Per quanto riguarda la sintassi, come precedentemente specificato, la struttura a blocchi indentati viene ripresa da Python. A causa di questa scelta il parser dev'essere messo a conoscenza della presenza degli a capo necessari a riconoscere l'inizio di un blocco e la terminazione di ogni istruzione (dal momento che non si fa uso del `';'`). Ciò ha portato a dover esplicitare in molte regole la presenza del token `NEWLINE` per evitare la creazione di errori di parsing dovuti ad a capo presenti nel testo in ingresso al solo scopo di aumentarne la leggibilità. Anche gli snippet di codice nativo, come gli a capo, non sono ignorati dal parser, in quanto è necessario poterli gestire semanticamente. Trattandoli esplicitamente è stato possibile, inoltre, definire in quali parti dello script si possono trovare.

Nel seguito sono riportate le regole sintattiche della grammatica. La notazione è la stessa di quella definita per i token. Le regole sintattiche hanno la prima lettera minuscola, le regole lessicali hanno la prima lettera maiuscola.

Come già anticipato uno script YARC è composto da un header di dichiarazione e tre macro-sezioni, di cui due opzionali:

```
scenario:
  NEWLINE*
  declaration
  (code_snippet | NEWLINE)*
  settings?
  stage
  writers?
  code_snippet*
  EOF
;

code_snippet : SNIPPET;
```

Nell'header vengono riportati il nome dello scenario e opzionalmente la libreria target verso cui tradurre lo script:

```
declaration : SCENARIO ID (COLON name)? NEWLINE;
```

Le tre macro-sezioni si dividono in:

1. `settings`, in cui vengono definite le variabili globali e i parametri di rendering (es. quante scene, fps, …) tramite semplici assegnamenti.

```
settings : SETTINGS COLON NEWLINE
  INDENT
    (setting | code_snippet)+
  DEDENT
;
setting  : ID ASSIGN test NEWLINE;
```

2. `stage`, in cui viene composta la scena e ne viene definita la dinamica.

```
stage : STAGE COLON NEWLINE INDENT stmts DEDENT;
stmts:
  open_stmt?
  (aug_expr_stmt | edit_stmt | behavior_stmt | code_snippet)+
;
```

3. `writers`, in cui vengono definite quali tipologie di dati sintetici generare (es. RGB, depth maps, bounding boxes, …). Anche i parametri dei writer sono gestiti tramite semplice assegnamento.

```
writers      : WRITERS COLON NEWLINE
  INDENT
    (expr_stmt | code_snippet | writer)+
  DEDENT
;
writer       : ID COLON NEWLINE INDENT writer_param+ DEDENT;
writer_param : ID ASSIGN test NEWLINE;
```

Uno stage può essere composto utilizzando vari statement:

- `open statement`, importa un modello 3D e lo usa come ambiente nel quale creare gli ulteriori oggetti.

```
open_stmt : OPEN test NEWLINE;
```

- `expression` e `augmented expression statements`, assegnano a una variabile il valore di un'espressione o i risultati di una ricerca di risorse al percorso specificato. Nel secondo caso sono permesse anche le `model expressions`, spiegate più avanti (in questo caso l'assegnamento a variabile si può omettere).

```
expr_stmt:
  namelist (AUG_ASSIGN | ASSIGN) (testlist | fetch_expr) NEWLINE
;
aug_expr_stmt: (
    namelist (
      AUG_ASSIGN (testlist | fetch_expr) NEWLINE
      | ASSIGN ( (testlist | fetch_expr) NEWLINE | model_expr)
    )
  )
  | model_expr
;

model_expr:
  create_expr | instantiate_expr | get_expr | group_expr
;

fetch_expr:
  FETCH test FROM test (MATCH test)? (LIMIT test)? RECURSIVE?
;
```

- `edit statement`, applica una serie di modifiche agli attributi di un oggetto. Può essere usato anche per manipolare la timeline

```
edit_stmt:
  EDIT (
    TIMELINE COLON NEWLINE INDENT (name test NEWLINE)+ DEDENT
    | test edit_block
  )
;
```

Le `model expressions` sono una serie di espressioni che definiscono la creazione, selezione e raggruppamento di modelli 3D all'interno di uno stage. Ognuna di queste può essere opzionalmente seguita da un blocco che specifica quali attributi settare e quale comportamento dare all'oggetto. La differenza tra `CREATE` e `INSTANTIATE` è la gestione degli oggetti: quando è necessario creare molti oggetti che servono solo da ambientazione o occlusioni si dovrebbe preferire `INSTANTIATE` per avere una gestione più efficiente da parte dell'engine, quando il modello da creare è uno degli elementi principali della scena si dovrebbe preferire `CREATE`.

```
create_expr:
  CREATE test? (
    (SHAPE | LIGHT) (edit_block | NEWLINE)
    | (STEREO CAMERA | CAMERA) (edit_block | NEWLINE)
    | FROM test (edit_block | NEWLINE)
    | MATERIAL (simple_block | NEWLINE)
  )
;
instantiate_expr : INSTANTIATE test? FROM test (edit_block | NEWLINE);
group_expr:
  GROUP LBRACK test (COMMA test)* RBRACK (edit_block | NEWLINE)
;
get_expr:
  GET (
    (CAMERA  | LIGHT | MATERIAL | ID ) AT
  )? test (simple_block | NEWLINE)
;

edit_block   : COLON NEWLINE INDENT (attr | inner_behavior_stmt)+ DEDENT;
simple_block : COLON NEWLINE INDENT simple_attr+ DEDENT;
attr         : core_attr | simple_attr | compound_attr;
```

All'interno dei blocchi si possono settare o modificare tre tipi di attributi:

1. Attributi Core, sono attributi che tutti gli oggetti hanno e quindi possono essere gestiti diversamente da tutti gli altri. Di norma i più frequenti.

   ```
   core_attr: (
       TRANSLATE AXIS? TO test
       | ROTATE AXIS? test
       | SCALE test
       | LOOK_AT test
       | UP_AXIS test
       | SIZE test
       | PIVOT test
       | SEMANTICS test
       | VISIBLE test
       | MATERIAL_ test
     ) NEWLINE
   ;
   ```

2. Attributi semplici, sono coppie chiave-valore (con opzionalmente il tipo in caso l'attributo non sia già presente nell'oggetto)

   ```
   simple_attr : name (COLON name)? test NEWLINE;
   ```

3. Attributi composti, definiscono comportamenti più complessi e spesso necessitano a loro volta di un blocco di attributi per settarne i parametri.

```
compound_attr: (
    SCATTER ON name (simple_block | NEWLINE)
    | ROT_AROUND name (simple_block | NEWLINE)
    | PHYSICS (simple_block | NEWLINE)
    | MOVE_TO_CAM name (simple_block | NEWLINE)
  )
;
```

L'ultimo statement riguarda invece la definizione del comportamento dinamico della scena. Ne esistono due versioni: quella interna a un blocco di modifica e quella top-level. La principale differenze è che nella versione interna tutti gli attributi fanno automaticamente riferimento all'oggetto a cui è relativo il blocco di modifica, nella versione top-level si possono invece istanziare oggetti 3D e definire il comportamento di più oggetti nello stesso statement.

```
behavior_stmt  : behavior_expr behavior_block;
behavior_expr  : EVERY test? (FRAMES | TIME);
behavior_block :
  COLON NEWLINE INDENT (aug_expr_stmt | code_snippet | edit_stmt)+
DEDENT;

inner_behavior_stmt  : behavior_expr inner_behavior_block;
inner_behavior_block : COLON NEWLINE INDENT attr+ DEDENT;
```

Infine, la sintassi delle espressioni è derivata da quella di Python, con minimi cambiamenti:

- Le tuple sono state rimosse e sono stati aggiunti i vettori, formati da tre valori e delimitati da `<...>`.
- Negli atomi sono stati aggiunti esplicitamente `len(...)` per indicare il numero di elementi di una collezione, le distribuzioni di probabilità (viste come chiamate a funzioni) e i range, definiti tra parentesi quadre, con la stessa semantica dell'operatore di indicizzazione degli array in Python.

```
test        : or_test (IF or_test ELSE test)?;
test_nocond : or_test;
or_test     : and_test (OR and_test)*;
and_test    : not_test (AND not_test)*;
not_test    : NOT not_test | comparison;
comparison  : expr (comp_op expr)*;
comp_op:
  LT
  | GT
  | EQUALS
  | GT_EQ
  | LT_EQ
  | NOT_EQ
  | IN
  | NOT IN
  | IS
  | IS NOT
;
expr       : xor_expr (BIT_OR xor_expr)*;
xor_expr   : and_expr (XOR and_expr)*;
and_expr   : shift_expr (BIT_AND shift_expr)*;
shift_expr : arith_expr ((LSHIFT | RSHIFT) arith_expr)*;
arith_expr : term ((PLUS | MINUS) term)*;
term       : factor ((MUL | DIV | MOD | IDIV) factor)*;
factor     : (PLUS | MINUS | BIT_NOT) factor | power;
power      : atom_expr (POWER factor)?;
atom_expr  : atom trailer*;
atom:
  (
    LPAREN test RPAREN
    | LBRACK testlist_comp? RBRACK
    | LT vector_comp? GT
    | LBRACE dict_or_set_maker? RBRACE
    | LEN LPAREN test RPAREN
    | name
    | SETTING_ID
    | distribution
    | INTEGER
    | FLOAT_NUMBER
    | STRING
    | NONE
    | TRUE
    | FALSE
  )
;
```

```
name : ID | UNDERSCORE;
distribution:
  DISTRIBUTION LPAREN arglist RPAREN
  | COMBINE LPAREN arglist RPAREN
;

testlist_comp :
  test (
    comp_for
    | (COMMA test)*
    | COLON test (COLON test)?
  )
;

vector_comp    : expr COMMA expr COMMA expr;
trailer        : LBRACK subscriptlist RBRACK | DOT name;
arglist        : argument (COMMA argument)*;
argument       : test (ASSIGN test)?;
subscriptlist : subscript_ (COMMA subscript_)*;
subscript_     : test (COLON test? sliceop?)? | COLON test? sliceop?;
sliceop        : COLON test?;

namelist : name (COMMA name)*;
testlist : test (COMMA test)*;
dict_or_set_maker:
  test (
    COLON test (comp_for | (COMMA test COLON test)*)
    | comp_for
    | (COMMA test)*
  )
;

comp_iter : comp_for | comp_if;
comp_for  : FOR namelist IN or_test comp_iter?;
comp_if   : IF test_nocond comp_iter?;
```

3 Analisi semantica

YARC è stato progettato per poter creare script portabili da una libreria a un'altra, semplicemente cambiando la riga di intestazione dello script. Ciò significa che deve poter cambiare il codice generato e i controlli da effettuare durante il parsing sulla base della libreria target. Dev'essere inoltre facile per gli sviluppatori aggiungere nuovi target senza impattare sul codice già presente.

Per ottenere questo comportamento, l'*handler semantico* viene inizializzato solamente dopo aver letto l'intestazione dello script (questo ne motiva la presenza obbligatoria come prima riga). Basandosi sulla libreria scelta, una *factory* inizializza l'handler corretto e setta, inoltre, il file di template per la traduzione dello script.

Il file di template è in formato `.stg` e viene utilizzato da StringTemplate 3 per generare testo, rimpiazzando dei placeholder con i valori correnti letti dal file in input. StringTemplate (ST) è un template engine sviluppato dallo stesso team di ANTLR e per questo si integra perfettamente con ANTLR 3. Per riscrivere una regola basta semplicemente estenderla aggiungendo il simbolo `->` seguito dal nome del template da usare, passandogli un lista di coppie chiave-valore, dove la chiave è il nome del placeholder e il valore è il valore da sostituire nel placeholder. Per esempio, la regola `open_stmt` si presenta nel seguente modo:

```
open_stmt : OPEN test NEWLINE -> open_stmt(path={$test.st});
```

dove il secondo `open_stmt` è il nome del template nel file `.stg`. Nella sua forma più semplice un template è una stringa e tra parentesi angolari si trovano i placeholder:

```
open_stmt(path) ::= <<
omni.usd.get_context().open_stage(<path>)
>>
```

Per la sintassi di ST3 e l'integrazione con ANTLR 3 si rimanda alla documentazione ufficiale[3].

La riscrittura di una regola si trova nel campo `st` dell'oggetto restituito dalla regola stessa. Le varie riscritture si costruiscono con un approccio bottom-up, il rewriting delle regole a livello più basso viene utilizzato nelle regole a livello più alto.

[3] https://theantlrguy.atlassian.net/wiki/spaces/ST/overview

L’handler semantico specifico per una libreria target si occupa di effettuare dei semplici controlli, di mappare parte del lessico di YARC rispetto alla libreria e di gestire errori e warning. In particolare, l’handler semantico si occupa di:

- Gestire i setting, definisce una collezione di setting di default e ne gestisce la riscrittura quando un certo setting dev’essere trattato diversamente (es. la risoluzione delle immagini in output).
- Gestire le variabili, definisce uno stack di symbol table per la definizione e lookup delle variabili dichiarate. Supporta anche il concetto di scope, secondo la filosofia dei classici linguaggi di programmazione, il lookup avviene prima nel blocco locale e poi iterativamente nei blocchi esterni. A causa della sintassi scelta a volte è necessario disabilitare temporaneamente il lookup delle variabili e riattivarlo solo successivamente, posticipando i vari controlli.
- Effettuare il parsing degli snippet di codice, per estrarne il codice e le variabili dichiarate (così che poi possano essere utilizzate dallo script senza generare errori), oltre che per verificarne la sintassi se possibile.
- Gestire i blocchi di attributi, dal momento che un blocco di attributi si può trovare in concomitanza di una creazione, alcuni possono essere passati al costruttore dell’oggetto 3D, in questo modo il codice generato risulta più compatto e di più facile comprensione; serve quindi separare gli attributi che possono essere passati al costruttore da quelli che vanno settati successivamente e dai comportamenti dinamici (quali attributi selezionare dipende da un dizionario, specifico per ogni libreria).
- *Espandere* le stringhe, per facilitare la scrittura degli script si è deciso di mettere a disposizione una variabile globale che definisce qual è il punto di `mount`, ossia in quale cartella si trovano tutte le risorse; in tutte le stringhe che iniziano con `'*/'`, il `*` viene sostituito con il valore presente nella variabile `mount`. Ad esempio, se `mount = '/home/user/Brakes'` e la variabile `brake_model = '*/brake.usd'`, questa nel codice generato in output viene espansa e varrà `brake_model = '/home/user/Brakes/brake.usd'`.

3.1 Gestione degli errori

Come già anticipato precedentemente, gli errori lessicali vengono gestiti sintatticamente. Gli errori sintattici sono gestiti tramite il metodo `displayRecognitionError` sovrascritto dal parser mentre gli errori semantici sono tutti gestiti dall'handler. L'handler semantico si occupa anche di formattare gli errori e fornire una messaggistica standard.

Ogni errore fa capo a una di sette tipologie (dettagliate in seguito) e ha un messaggio che ne motiva la causa, il punto in cui è stato generato, e per facilitarne la ricerca, il token che l'ha generato viene evidenziato. L'idea è quella di mostrare all'utente in modo molto intuitivo dove si trova l'errore e come rimuoverlo.

Le tipologie di errore si ispirano nuovamente a Python, viste le strutture sintattiche simili e sono le seguenti:

- **SyntaxError**: un tipo di errore generico, causato da un'errata sintassi (es. mancanza di un token o presenza di un token non riconosciuto, errore lessicale).
- **IndentationError**: riguarda tutti gli errori relativi al livello di indentazione, ha tre cause:
    - Aumento del livello di indentazione quando non richiesto.
    - Diminuzione del livello di indentazione senza una corrispondenza con i livelli precedenti.
    - Assenza di indentazione quando richiesta da un blocco.
- **LibraryError**: generato esclusivamente dalla factory quando la library richiesta non è tra quelle supportate.
- **SnippetError**: generato quando il parsing degli snippet non termina correttamente a causa di un errore sintattico presente nel codice.
- **SettingError**: generato quando si fa riferimento a una variabile globale inesistente.
- **WriterError**: generato quando il writer selezionato non è tra quelli supportati della libreria scelta
- **NameError**: riguarda tutti gli errori relativi alle variabili, ha due cause:
    - Viene utilizzata una variabile non dichiarata
    - Il nome di una variabile coincide con una keyword in codice nativo

3.2 Gestione dei warning

Mentre gli errori sono tutti relativi a casistiche in cui la sintassi è errata oppure a casi in cui il codice prodotto in output porterebbe a errori (es. nomi delle variabili che coincidono con keyword o non dichiarate), i warning riflettono comportamenti *sconsigliati* ma che non impattano sulla correttezza del codice prodotto. Per questo motivo è possibile disattivarne il controllo su richiesta.

Come gli errori, anche i warning hanno un tipo e un messaggio, anche in questo caso con il punto in cui è stato generato e la causa. Al contrario degli errori invece, i warning sono completamente generati e gestiti dall'handler semantico.

Le tipologie di warning sono le seguenti:

- **UnusedVariable**: generato quando una variabile viene dichiarata ma non viene mai utilizzata nello script.
- **UnknownParameter**: generato quando nel blocco di modifica ci sono attributi che l'handler non sa come gestire e ignora. In questo caso viene anche suggerito l'attributo con il nome più simili, in modo da aiutare l'utente in caso abbia commesso qualche errore nello scrivere lo script.
- **UnsupportedTarget**: 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 modifica vengono modificati più volte gli stessi attributi (es. settando `scale` e `size`, oppure `rotate` e `look_at`). In questo caso vengono mostrati tutti gli attributi che sono stati sovrascritti e quale attributo ha effetto sull'oggetto.
- **MissingWriterParameter**:generato quando alcuni attributi dei writer non sono stati specificati ma possono essere settati di default (es. il percorso in cui salvare i dati di ogni writer).

3.3 Compiler

Per semplificare l'inizializzazione di parser e lexer viene fornita la classe `YarcCompiler`. Questa si occupa di creare il lexer, creare lo stream di token e passarlo al parser.

Permette inoltre di settare la libreria e il valore di alcune variabili globali, sovrascrivendo i valori presenti all'interno dello script; così facendo è più semplice rigenerare l'output qualora servisse cambiare solo il numero di scene o il punto di mount, senza dover necessariamente mettere mano allo script.

Questa classe è stata pensata per poter usare YARC programmaticamente all'interno di altre applicazioni o librerie, fornendo un'interfaccia pulita.

4 CLI Tool

È possibile utilizzare YARC anche standalone, facendo uso dello strumento a linea di comando.

Il tool è stato sviluppato sfruttando Typer e Rich. La prima è una libreria che agevola lo sviluppo di applicazioni a riga di comando, la seconda è una libreria che fornisce strumenti per migliorare l'output e la visualizzazione di testo e dati nel terminale, come syntax highlighting e progress bar.

Lo strumento fornisce due funzionalità:

1. La prima, invocata tramite il comando `compile` seguito da uno script YARC, effettua la traduzione in codice nativo. È possibile passare come opzioni: la libreria, il punto di mount, il numero di scene e se generare i warning. Se non sono presenti errori mostra il codice a terminale oppure lo salva al percorso indicato tramite l'apposita opzione. In caso di errore, mostra tutti gli errori incontrati e non produce alcun codice in output.
2. La seconda, invocata tramite il comando `translate-grammar` traduce la grammatica ANTLR in uno degli altri target supportati da ANTLR 3, per esempio Java o C++. Questo permette di limitare lo sforzo degli sviluppatori quando si renderà necessario migrare verso altri linguaggi target qualora si volesse integrare YARC in un nuovo engine 3D non compatibile con Python. La grammatica generata mantiene tutte le regole e i rewriting, vengono però adattate le azioni semantiche (per esempio `this` al posto di `self`).

5 Per sviluppatori

YARC è sviluppato in Python 3 e si è scelto di utilizzare Poetry come gestore delle dipendenze.

Sono stati settati dei pre-commit hooks per favorire la qualità del codice e facilitare il linting e la formattazione del codice. È inoltre presente un Makefile con alcuni comandi per velocizzare lo sviluppo, come l'update delle dipendenze, l'esecuzione dei test e il linting.

La grammatica completa di azioni semantiche si può trovare nella stessa repository[4] del codice sorgente.

5.1 Tecnologie e strumenti usati

- Python 3.10 e Virtual Environments (modulo `venv`).
- Poetry: gestore delle dipendenze e del packaging di un'applicazione Python.
- cookie-cutter: strumento per la creazioni di progetti Python a partire da template.
- ANTLR 3: libreria per la generazione di lexer e parser a partire da una file di grammatica.
- StringTemplate 3: libreria per la definizione e istanziazione di template per la generazione testuale.
- Typer: libreria per la prototipazione rapida di applicazioni da linea di comando.
- Rich: libreria per la formattazione di testo e strutture dati nel terminale.
- Pre-commit: strumento per automatizzare il linting e formattazione del codice, lavora per mezzo di *hooks*. Quelli impostati sono:
    - pyupgrade, aggiornamento della sintassi per pattern sconsigliati/deprecati.
    - autoflake, rimozione di import e variabili non utilizzati.
    - isort, riordinamento degli statement di import secondo standard.
    - black, formattazione del codice secondo standard.
- pytest e pytest-cov: librerie rispettivamente per il testing e il monitoraggio del coverage del codice.
- mypy: libreria per il type checking in Python.

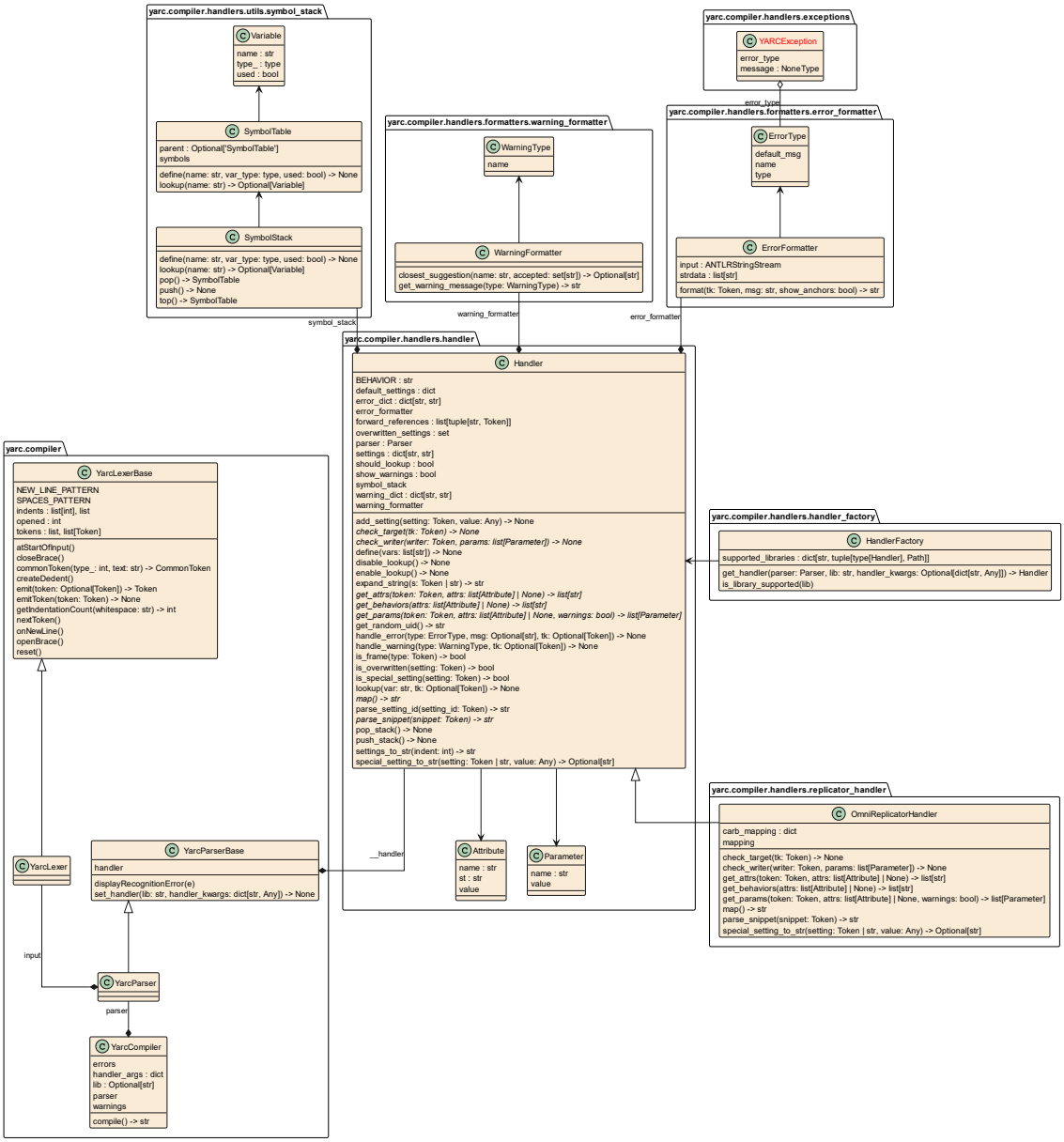
[4] https://github.com/fcarne/yarc

- GitHub e GitHub Actions: versioning del codice e pipeline di CI/CD per il testing.
- Make: tool per la creazione di script di utility per automatizzare e semplificare alcuni dei task di gestione, building e testing del codice.
- Visual Studio Code: Text editor per lo sviluppo del codice con le estensioni necessarie per semplificare lo sviluppo dei file di specifica per ANTLR e StringTemplate.

5.2 Diagrammi UML

Nel seguito è riportato il class diagram di YARC. Sono stati esclusi i moduli legati al tool da riga di comando e i metodi di `YarcParser` e `YarcLexer` auto-generati da ANTLR. Oltre al codice sorgente, ci sono due cartelle contenti file di specifica:

- `grammar`, che contiene la grammatica ANTLR che definisce lexer e parser.
- `templates`, che contiene i template per ogni handler supportato.



5.3 Aggiungere una nuova libreria target

Qualora si volesse estendere la compatibilità di YARC ad altre librerie sono necessari tre step:

1. Generare un handler semantico specifico per la libreria, ereditando dalla classe astratta `Handler`.
2. Creare il file `.stg` con i template per la nuova libreria.
3. Modificare la factory in modo tale da metterla a conoscenza della presenza del nuovo handler.

5.4 Integrare YARC con un engine

Se si vuole integrare YARC con un engine che supporta Python si può utilizzare la libreria *as is*, estendendola se necessario. È consigliato utilizzare il `YarcCompiler`, evitando se possibile di usare direttamente parser e lexer.

Se l'engine non supporta Python, parser e lexer compatibili con il linguaggio di programmazione supportato possono essere facilmente derivati utilizzando il tool da riga di comando.