Yass - Documentazione tecnica

Xhorxho Papallazi, Marco Rodolfi, Simone Ronzoni

21 febbraio 2024

1 Introduzione

Yass è un linguaggo di stile che estende il CSS andando ad aggiungere funzionalità quali variabili, cicli, annidamenti e mixin.

CSS (acronimo di Cascading Style Sheets) è un linguaggio usato per definire la formattazione e lo stile visuale di documenti HTML, XHTML o XML.

Fornisce un controllo completo sulla presentazione di pagine web. Le regole per costruire un CSS sono contenute in un insieme di direttive (le recommendations) emanate a partire dal 1996 dal W3C.

Un file CSS è costituito da:

- *regole* che sono coppie costituite da un selettore e un blocco di dichiarazioni, racchiuso tra parentesi graffe;
- *selettore* è un predicato che individua certi elementi del documento HTML;
- *dichiarazioni*, separate con un punto e virgola dalle altre, e a loro volta costituite da una proprietà, ovvero un tratto di stile (come il colore del testo) e un valore da assegnare a quest'ultimo (per esempio blu), separati dai due punti.

Yass mantiene la stessa struttura di regole del CSS ma ne estende le funzionalità per renderlo più potente.

I termini a cui si farà riferimento nel documento derivano dal CSS stesso.

2 Requisiti

Tramite Yass si vogliono implementare dei costrutti tipici dei linguaggi di programmazione ed adattarli ad un utilizzo per la definizione di stili. In questo paragrafo vengono definiti i vari requisiti che ci si era prefissati di rispettare, ossia di permettere di:

- definire variabili e gestirne lo scoping
- definire cicli e permetterne gli annidamenti
- rendere dinamici i selettori, le proprietà e i loro valori
- annidare classi CSS
- definire delle funzioni, chiamate *mixin*

2.1 Variabili

È possibile definire delle variabili da utilizzare sia per specificare nomi di proprietà, valori di proprietà, identificativi per poter essere riutilizzati (ad esempio come valore dei selettori delle classi).

Le variabili devono essere necessariamente definite prima del loro utilizzo e la loro tipizzazione è dinamica.

Una variabile non può essere riassegnata.

Sono tre i tipi di variabili supportati:

1. Stringhe
2. Liste
3. Dizionari

È possibile assegnare ad una variabile il valore di un'altra variabile in fase di definizione della variabile stessa.

In base al contesto in cui ci si trova (ad esempio definizione di variabili o loro utilizzo) vengono effettuati controlli sull'esistenza delle variabili, sul loro tipo o altri controlli. Per una lista dei possibili errori fare riferimento al paragrafo 6.7.

Si riporta di seguito un esempio di utilizzo delle variabili.

Yass

```
bg-color = "blue";

body .${bg-color} {
  background-color: ${bg-color};
}
```

CSS

```
body .blue {
  background-color: blue;
}
```

2.2 Cicli

I cicli permettono di iterare su variabili di tipo lista e dizionario (gli unici tipi iterabili), e per ogni elemento vengono generate le regole definite nel corpo del ciclo.

Sono disponibili due sintassi per definire i cicli:

1. `$foreach(<iterable variable>)` accetta come parametro una variabile iterabile e predispone nello scope delle variabili locali del ciclo la variabile `value` che contiene il valore e `index` l'indice dell'elemento i-esimo della variabile iterabile;

2. `$foreach(<index name>, <item name> : <iterable variable>)` è necessaria nel caso di cicli annidati per ridefinire i nomi delle variabili definite nello scope delle variabili locali del ciclo con una sintassi simile a quella di Java. Il primo identificativo consentirà l'accesso alla chiave, in caso di dizionari, o all'indice, in caso di liste, dell'iterazione corrente. Il secondo identificativo si riferirà al valore all'iterazione corrente e dopo il carattere `:` viene specificata la variabile iterabile.

Sono supportati cicli annidati e la possibilità di richiamare mixin e variabili definite esternamente.

Di seguito viene riportato un esempio di definizione di un ciclo con la prima sintassi.

Yass

```
bg-colors = ["blue", "white",
↪  "green", "yellow", "black"];

$foreach(bg-colors) {
  body .${value} {
    background-color: ${value};
  }
}
```

CSS

```
body .blue {
    background-color: blue;
}

body .white {
    background-color: white;
}

body .green {
    background-color: green;
}

body .yellow {
    background-color: yellow;
}

body .black {
    background-color: black;
}
```

Di seguito viene riportato un esempio di definizione di un ciclo con la seconda sintassi.

Yass

```
bg-colors = ["blue", "white",
↪  "green", "yellow", "black"];
$foreach(index, color : bg-colors) {
  body .color-${index} {
    background-color: ${color};
  }
}
```

CSS

```
body .color-0 {
    background-color: blue;
}

body .color-1 {
    background-color: white;
}

body .color-2 {
    background-color: green;
}

body .color-3 {
    background-color: yellow;
}

body .color-4 {
    background-color: black;
}
```

2.3 Annidamento di regole CSS

Questa funzionalità permette di specificare stili annidati andando a definire all'interno di un blocco CSS altre regole CSS. Il risultato sarà la creazione di regole CSS che hanno come selettore la concatenazione dei selettori padre con quelli del figlio.

È stato introdotto l'operatore & per riferirsi ai selettori della classe padre, potendo specificare la posizione in cui inserirli, (ad esempio per generare da classi "base", come `btn`, classi specifiche, come `btn-primary`, o pseudo-classi, come `btn:hover`).

Yass

```
body {
  font-size: 1rem;
  font-family: Arial;

  &:hover {
    cursor: pointer;
  }

  h1 {
    font-size: 3rem;
  }
}
```

CSS

```
body {
    font-size: 1rem;
    font-family: Arial;
}

body:hover {
    cursor: pointer;
}

body h1 {
    font-size: 3rem;
}
```

2.4 Mixin

I mixin permettono di definire delle proprietà CSS riutilizzabili, permettendo di parametrizzare determinati valori.

Sono in qualche misura l'analogo delle funzioni dei normali linguaggi di programmazione permettendo la definizione di una struttura richiamabile con parametri diversi.

Yass

```
button = (bg-color) {
  padding: 12px;
  background-color: ${bg-color};
};

button-primary-color = #2236e1;
button-secondary-color = #0a1043;

button .btn-primary {
  $button(button-primary-color);
  font-size: 1.5rem;
}

button .btn-secondary {
  $button(button-secondary-color);
}
```

CSS

```
button .btn-primary {
  padding: 12px;
  background-color: #2236e1:
  font-size: 1.5rem;
}

button .btn-secondary {
  padding: 12px;
  background-color: #0a1043;
}
```

3 Tecnologie utilizzate

Per la realizzazione di Yass è stato utilizzato ANTLRv3 [4] per la costruzione del linguaggio con linguaggio di output Java [3].

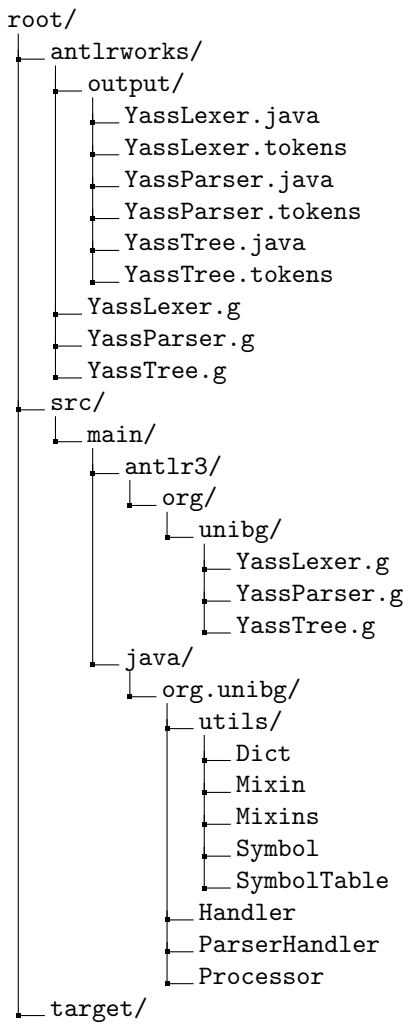
Dopo una prima fase di ricerca sulla documentazione di ANTLR si è scelto di creare una rappresentazione Abstract Syntax Tree (AST) sia per provare questa funzionalità dello strumento, che per la facilità di comprensione che offre questa rappresentazione.

Il parser ANTLR riconosce gli elementi presenti nel codice, scritto nel linguaggio definito, e costruisce un *parse tree*. Dal parse tree si ottiene poi un AST che verrà utilizzato per eseguire la validazione e produrre il codice compilato/tradotto. ANTLR considera anche il parse tree come un AST ma è utile considerare il parse tree come l'albero che rappresenta l'informazione del il parser mentre l'AST contiene l'informazione riorganizzata per facilitare i passi successivi.

Per la traduzione dell'albero generato dal parser in codice CSS si è deciso di integrare codice Java all'interno del file `.tree` e non utilizzare StringTemplate [6], che era stata la soluzione scelta inizialmente perché reputata una soluzione interessante e molto potente, per difficoltà legate all'implementazione dei cicli.

4 Struttura del progetto

La struttura di cartelle del progetto è la seguente:



Si è deciso di dividere la grammatica di ANTLR in tre file che contengono:

1. `YassLexer.g` il lexer;
2. `YassParser.g` il parser;
3. `YassTree.g` l'albero della grammatica decorata.

Il file con estensione `.tree` è dovuto a ciò che è stato specificato nel paragrafo 3, dove è stato spiegato che si è creata una rappresentazione intermedia della grammatica sotto forma di Abstract Syntax Tree (AST) che poi è stato visitato per generare la traduzione in CSS.

Nella cartella `antlrworks` sono presenti i file `.g` di ANTLR, mentre in `output` i token e file Java generati utilizzando ANTLRWorks [5].

Nella cartella `src` sono contenuti i package e le classi Java.

Nella cartella `target` è contenuto il risultato della compilazione con Maven [7].

5 Compilazione del progetto

Come ambiente di sviluppo è stato utilizzato IntelliJ IDEA [1] con il plugin per Maven [7] che viene utilizzato come strumento di automazione di build utilizzando come archetipo quello per progetti che fanno uso di ANTLRv3 [4] [2].

Nell'interfaccia di IDEA si può trovare la sezione dedicata al plugin di Maven a destra dello schermo come si può vedere nella figura 1. Premendo sopra il pulsante si aprirà l'interfaccia dedicata con le varie operazioni permesse:

- `clean`: elimina la cartella `target`
- `compile`: compila il progetto generando la cartella `target`
- `package`: crea all'interno della cartella `target` due file `jar` che contengono le classi compilate e uno dei due anche tutti i pacchetti in un unico bundle jar.

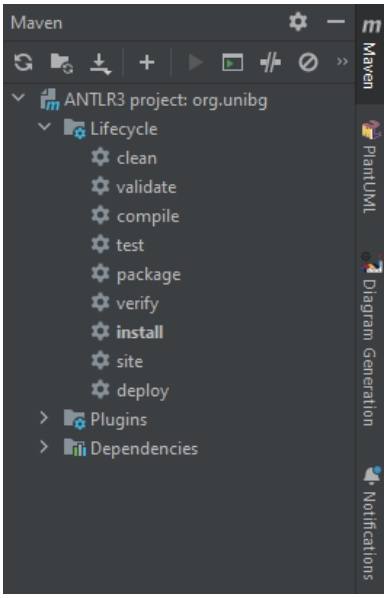


Figura 1: Schermata di build di Maven nell'interfaccia di IDEA

6 Dettagli implementativi

In questo paragrafo viene mostrata la grammatica non decorata del linguaggio e sono descritti alcuni dettagli implementativi con le rispettive strategie utilizzate.

Seguendo le varie sezioni e i file a cui si riferiscono si possono comprendere le note.

6.1 Grammatica

Come già specificato la grammatica viene utilizzata per generare un AST da cui poi verrà generato il codice CSS. È da notare quindi l'opzione `output = AST;` impostata nell'intestazione della grammatica visibile nel codice 1.

Listing 1: Grammatica non decorata

```
parser grammar YassParser;

options {
  output = AST;
  tokenVocab = YassLexer;
  backtrack = true;
}

tokens {
        RULE;
        BLOCK;
        PROPERTY;
        ATTRIB;
        SPACEDELEMENT;
        ELEMENT;
        PSEUDO;
        VAR;
        INTERPOLATION;
        ASSIGNMENT;
        LIST;
        FOREACH;
        FOREACHBODY;
        DICT;
        DICTITEM;
        MIXIN;
        MIXINBODY;
        MIXINCALL;
        ITERGET;
}

@header {
package org.unibg;
}

@members {
ParserHandler ph;

public ParserHandler getHandler(){
        return ph;
}


public void displayRecognitionError(String[] tokenNames, RecognitionException e){
        String hdr = " * " + getErrorHeader(e);
        String msg = " - " + getErrorMessage(e, tokenNames);
```

```
        Token tk = input.LT(1);

        ph.handleError(tk, hdr, msg);
}


void initHandler(){
        ph = new ParserHandler(input);
}
}

// This is the "start rule".
stylesheet
        @init
        {
         initHandler();
        }
  : statement*
  ;

statement
  : ruleset
  | variableDeclaration terminator -> variableDeclaration
  | foreach
  ;

terminator
        : SEMI
        ;

// Variables
variableInterpolation
        : DOLLAR BlockStart Identifier BlockEnd -> ^(INTERPOLATION Identifier)
        ;

variableDeclaration
        : Identifier EQ variableValue -> ^(VAR Identifier variableValue)
        ;

identifier
        : variableInterpolation
        | get
        | Identifier
        ;

variableValue
        : variableAtom
        | list
        | dict
        | mixin
        ;

variableAtom
        : StringLiteral
        | Color
        | measurement
        | variableInterpolation
        | get
        ;

list
        : LBRACK variableAtom (COMMA variableAtom)* RBRACK -> ^(LIST variableAtom+)
```

```
        ;

dict
        : BlockStart dictItem (COMMA dictItem)* BlockEnd -> ^(DICT dictItem+)
        ;

dictItem
        : StringLiteral COLON variableAtom -> ^(DICTITEM StringLiteral variableAtom)
        ;

mixin
        : LPAREN (Identifier (COMMA Identifier)*)? RPAREN BlockStart mixinBody BlockEnd -> ^(
            ↪ MIXIN Identifier* mixinBody)
        ;

mixinBody
        : property+ -> ^(MIXINBODY property+)
        ;

// Iterable get
get
        : GET LPAREN Identifier COMMA StringLiteral RPAREN -> ^(ITERGET Identifier
            ↪ StringLiteral)
        | GET LPAREN Identifier COMMA Number RPAREN -> ^(ITERGET Identifier Number)
        ;

// For loop
foreach
        : FOR LPAREN Identifier RPAREN BlockStart foreachBody BlockEnd -> ^(FOREACH Identifier
            ↪  foreachBody)
        | FOR LPAREN Identifier COMMA Identifier COLON Identifier RPAREN BlockStart
            ↪ foreachBody BlockEnd-> ^(FOREACH Identifier Identifier Identifier foreachBody)
        ;

foreachBody
        : block -> ^(FOREACHBODY block)
        ;

// Mixins
mixinCall
        : Mixin LPAREN (Identifier (COMMA Identifier)*)? RPAREN terminator -> ^(MIXINCALL
            ↪ Mixin Identifier*)
        ;

// Style blocks
ruleset
    : selectors BlockStart block BlockEnd -> ^(RULE selectors block)
        ;

block
        : (property | ruleset | mixinCall | foreach)* -> ^(BLOCK property* mixinCall* foreach*
            ↪  ruleset*)
        ;

// Selector
selectors
        : selector (COMMA selector)*
        ;

// attrib* pseudo*
selector
        : nextElement+ attrib* pseudo? -> nextElement+ attrib* pseudo*
```

```
        ;

nextElement
        : element
                -> {ph.checkNextIsSpace()}? ^(SPACEDELEMENT element)
                -> ^(ELEMENT element)
        ;

// Element
element
        : selectorPrefix identifier
        | identifier
        | DOT identifier
        | HASH identifier
        | TIMES
        | PARENTREF
        | pseudo
        ;

selectorPrefix
  : GT
  | PLUS
  | TIL
  ;

// Attribute
attrib
        : LBRACK identifier (attribRelate (StringLiteral | identifier))? RBRACK -> ^(ATTRIB
            ↪ identifier (attribRelate StringLiteral* identifier*)?)
        ;

attribRelate
        : EQ
        | TILD_EQ
        | PIPE_EQ
        ;

// Pseudo
pseudo
        : (COLON|COLONCOLON) identifier -> ^(PSEUDO COLON* COLONCOLON* identifier)
        ;

// Properties
property
        : identifier COLON args terminator -> ^(PROPERTY identifier args)
        ;

args
        : expr (COMMA? expr)*
        ;

expr
        : measurement IMPORTANT? -> measurement IMPORTANT*
        | identifier IMPORTANT? -> identifier IMPORTANT*
        | Color IMPORTANT? -> Color IMPORTANT*
        | StringLiteral IMPORTANT? -> StringLiteral IMPORTANT*
        ;

measurement
  : Number Unit?
  ;
```

6.2 ParserHandler

Nella classe `ParserHandler.java` sono presenti la gestione degli errori per il lexer e il parser e una funzione di utilità `checkNextIsSpace()` che controlla se il prossimo token nascosto rappresenta il carattere di spazio. Questo serve perché lo spazio è normalmente nel canale nascosto ma nella sintassi del CSS si può definire una gerarchia di elementi all'interno di un selettore usando il carattere di spazio. Solo in questo caso quindi lo spazio deve essere considerato e viene quindi aggiunto come nodo nell'AST.

6.3 Handler

Nella classe `Handler.java` sono presenti le funzioni per la traduzione in CSS a partire dall'AST. Si riportano di seguito alcune considerazioni particolari:

- la stringa dei selettori è costituita da uno o più selettori separati da una virgola. Per poter permettere l'annidamento, un selettore potrebbe contenere il simbolo `&`: in questo caso, in fase di costruzione della stringa dei selettori, viene prefisso il valore il ritornato dal metodo `getCurrentSelector()`, che ritorna il selettore della regola padre.
- nell'attributo `level` è contenuto il livello di profondità a cui ci si trova, necessario poi per gestire gli annidamenti.
- alla chiusura del blocco di definizioni di una regola, viene concatenata la traduzione attuale con i blocchi del livello più alto.

6.4 Variabili

Le variabili sono gestite mediante l'oggetto di tipo `SymbolTable`, al cui interno sono memorizzate le variabili in una `HashMap`. In fase di dichiarazione di una variabile, viene creata un nuovo elemento nella HashMap con chiave il nome della variabile e come valore un oggetto `Symbol` contenente il tipo della variabile (stringa, lista o dizionario) e il rispettivo valore.

Il tipo stringa è il tipo base di ogni variabile, anche dei numeri e quindi non sono permesse operazioni aritmetiche.

Non è permesso cambiare il valore di una variabile quindi prima di inserire una variabile nella `SymbolTable` viene controllato che non sia già stata definita.

Per quanto riguarda variabili di tipo stringa, è definita la regola `variableInterpolation` per l'AST che permette l'interpolazione solo di variabili di tipo stringa andando a chiamare il metodo `getVarValue()` che va a prendere la variabile dalla SymbolTable, verifica la correttezza del tipo e ne restituisce il valore.

Per quanto riguarda liste e dizionari, anche questi sono memorizzati nella SymbolTable. Quando viene incontrata la regola `get` nell'AST per ottenere l'elemento i-esimo, viene richiamato il metodo `getSpecificValue()` con parametri la lista e l'indice o il dizionario e la chiave. Viene controllato che se l'elemento passato come parametro è una variabile di tipo lista allora il secondo parametro deve essere un valore intero, altrimenti se il primo è un dizionario, il secondo parametro deve essere una stringa.

6.5 Scope delle variabili

Come nella maggior parte dei linguaggi le variabili hanno uno scope limitato ai blocchi in cui vengono definite (ad esempio i blocchi iterazione o corpi di regole): quando il blocco termina la variabile viene eliminata.

Per supportare lo scoping delle variabili, quando viene soddisfatta la regola AST di cicli e mixin, viene creata una nuova SymbolTable con un riferimento alla SymbolTable padre. In questo modo, nel momento in cui si utilizzano cicli e mixin annidati, per ognuno verrà creato uno scope di variabili. In

fase di ricerca del valore, se una determinata variabile non è stata trovata nella SymbolTable corrente, verrà ricercata ricorsivamente nelle SymbolTable padre finché non si raggiungere la radice e viene eventualmente lanciato un errore.

Questo meccanismo ha permesso la definizione di variabili predefinite all'interno di cicli (come la variabile `value` per accedere al valore i-esimo) e la definizione di parametri nei mixin mascherando eventuali variabili definite esternamente con lo stesso nome senza cambiarne il valore.

In questo momento le regole CSS non costituiscono un blocco con uno scope, le variabili possono essere solo definite in uno scope "globale" oppure si possono usare le variabili locali definite nei mixin (che hanno il nome dei parametri di definizione del mixin) oppure nei cicli (con le sintassi mostrate nel sottoparagrafo 2.2).

6.6 Cicli

Esistono due sintassi per definire i cicli mostrate nel sottoparagrafo 2.2.

Viene effettuato un controllo per verificare che la variabile passata come argomento sia iterabile (lista o dizionario). La logica vera e propria è contenuta nel metodo `foreachBody()`.

Non potendo effettuare cicli durante la visita dell'AST, in quanto impossibile con l'implementazione base del parser dell'albero offerta da ANTLR, viene inizializzato un nuovo oggetto di tipo `TreeParser` (generato da ANTLR in automatico e che contiene le regole per visitare l'albero) a partire dal sotto albero in modo iterativo. È stata l'unica soluzione trovata, ANTLRv4 sembra facilitare questa funzionalità.

6.7 Gestione errori

Per la gestione degli errori semantici, è stato predisposto il metodo `handleError()` che prende come parametro l'errore di tipo `enum` e il `token` che ha originato l'errore, genera il messaggio di errore e lo aggiunge all'attributo `errorList` che viene mostrato in caso ci fossero errori durante la compilazione.

Gli errori gestiti sono i seguenti:

- `UNDECLARED_VAR_ERROR`: si verifica quando si tenta ad accedere a una variabile non preventivamente inizializzata.
- `DECLARED_VAR_ERROR`: si sta tentando di definire una variabile già definita.
- `NOT_ITERABLE_VAR_ERROR`: la variabile passata come argomento di un ciclo non è di tipo iterable (LIST o MAP).
- `DECLARED_MIXIN_ERROR`: Mixin già dichiarato.
- `UNDECLARED_MIXIN_ERROR`: Mixin non dichiarato.
- `NULL_VAR_ERROR`: La variabile è presente nella SymbolTable ma ha valore null.
- `MISMATCH_ARGUMENTS_MIXIN_ERROR`: Il numero di parametri passati ad un mixin non corrispondono a quelli dichiarati.
- `NOT_STRING_VAR_ERROR`: Quando la variabile che si vuole ottenere non ha il tipo corrispondente a quello richiesto.
- `INDEX_OUT_OF_RANGE_ERROR`: Si sta tentando di accedere a un elemento i-esimo di indice superiore alla lunghezza della struttura dati.

6.8 Processor

La classe `Processor` contiene la definizione dell'interfaccia a linea di comando (CLI) esposta per potersi interfacciare con il programma e i metodi per richiamare le funzionalità di ANTLR.

7 Diagramma di classe

Nell'immagine 2 viene riportato il diagramma di classe per il package `unibg`.

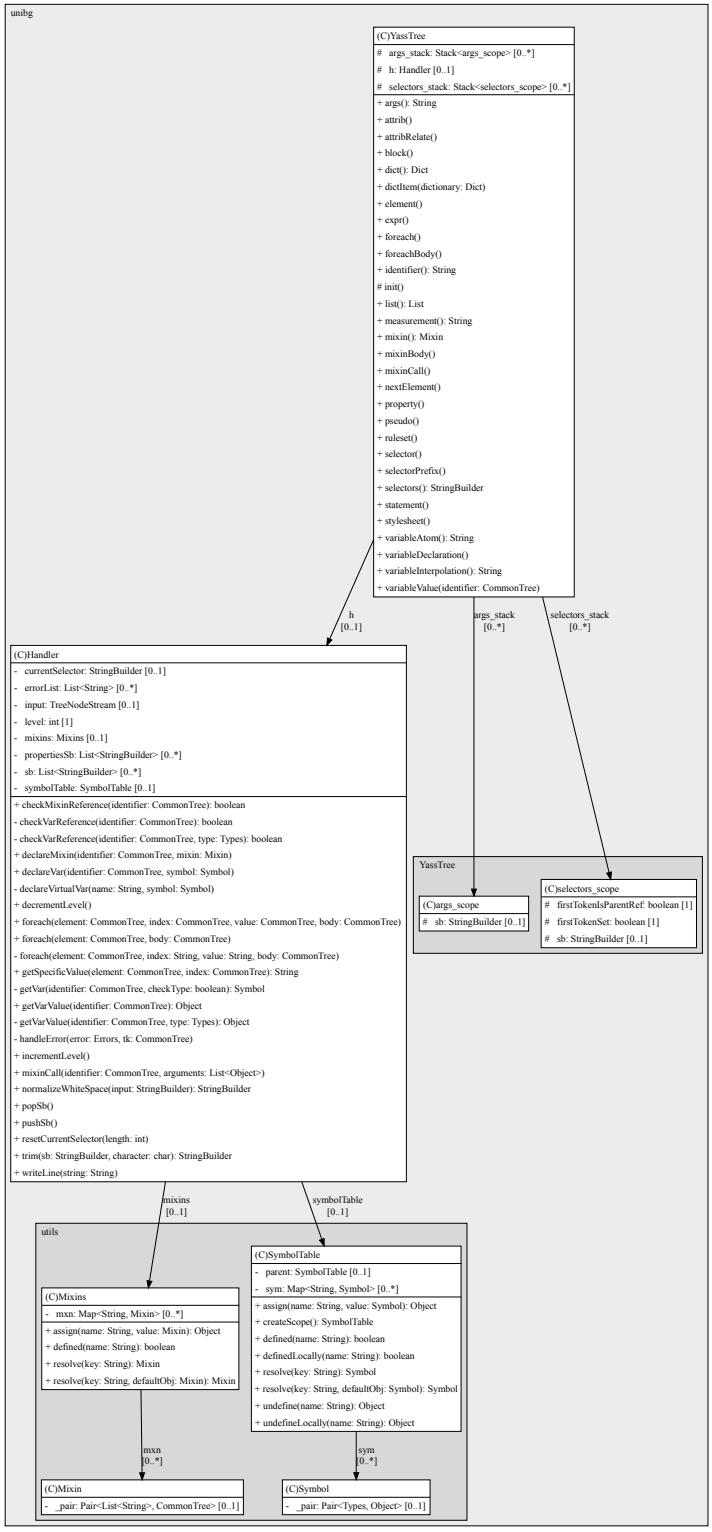


Figura 2: Diagramma di classe per il package unibg

Riferimenti bibliografici

[1] JetBrains s.r.o. Intellij idea. https://www.jetbrains.com/idea/. Visitato il: 2024-02-19.

[2] MvnRepository. Antlr 3 maven archetype. https://mvnrepository.com/artifact/org.antlr/antlr3-maven-archetype. Visitato il: 2024-02-19.

[3] Oracle. Java. https://www.java.com/en/. Visitato il: 2024-02-19.

[4] Terence Parr. Antlr v3. https://www.antlr3.org/download.html. Visitato il: 2024-02-19.

[5] Terence Parr. Antlrworks. https://www.antlr3.org/works/. Visitato il: 2024-02-19.

[6] Terence Parr. Stringtemplate. https://www.stringtemplate.org/. Visitato il: 2024-02-19.

[7] The Apache Software Foundation. Maven. https://maven.apache.org/. Visitato il: 2024-02-19.