# ANTLR + Java : Compilatore per il linguaggio **Simple Plus**

Alessio Portaro (0000000000)[1] and Andrea Ercolessi (0000000000)[1]

[1] *Laurea Magistrale in Informatica , Alma Mater Studiorum, Campus di Bologna*

Anno Accademico 2020-2021

# Indice

# 1 Introduzione

Vogliamo scrivere un compilatore per il linguaggio **Simple Plus (SP)** definito dalla grammatica scritta nel formalismo del tool *ANTLR*. A partire dal Lexer ed il Parser generati da *ANTLR* vogliamo definire nel linguaggio *JAVA* :

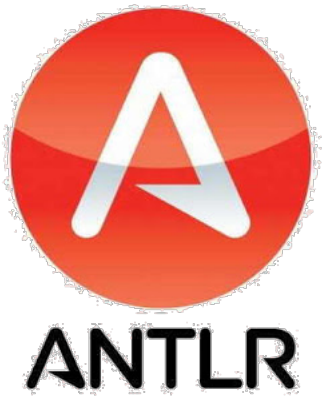
- Un'implementazione della Classe Visitor, che costruisca un ***Abstract Syntax Tree(AST)***
- Un analizzatore semantico che controlli che ogni variabile e funzione sia stata dichiarata, che non esistano dichiarazioni multiple nello stesso scope e la correttezza dei tipi rispetto al loro utilizzo.
  Inoltre l'analizzatore deve controllare gli ***Effetti*** che ogni operazione produce sulle variabili e soprattutto che l'ordine con cui questi *Effetti* si susseguono sia coerente rispetto al comportamento atteso.
- Un compilatore che traduca il codice ad alto livello in una serie di istruzioni bytecode precedentemente definite. Questo codice bytecode dovrà poi essere testato scrivendo una ***Virtual machine*** che definisca il comportamento di ogni istruzione e che poi possa eseguire il file ottenuto dalla compilazione

# 2 Strumenti e design

## 2.1 Tools utilizzati

Durante la fase di sviluppo sono stati utilizzati i seguenti strumenti :

- **ANTLR** : Acronimo di *ANother Tool for Language Recognition*, ANTLR è uno strumento per la generazione automatica di Parser per grammatiche LL. La versione utilizzata è la 4.6




- **Eclipse** : Come ambiente di sviluppo *JAVA* è stato utilizzato l'IDE Eclipse nella sua versione 03-2020

## 2.2 Struttura delle classi

Il progetto si divide in 4 package principali :

- **AST** : In questo pacchetto sono contenute tutte classi che descrivono i vari nodi dell'albero di sintassi astratto. Quest'ultimo è a sua volta una sintesi del *Parse tree* generato automaticamente da *ANTLR*. Ognuna di queste classi contiene principalmente 3 metodi :

  1. **CheckSemantics** : Questo metodo viene invocato quando il compilatore sta effettuando una discesa dell'albero di sintassi astratto, per effettuare un controllo sulla coerenza dei tipi e sull'esistenza di ogni variabile utilizzata, nello scope corrente
  2. **CheckEffects** : Questo metodo viene invocato quando il compilatore sta effettuando una discesa dell'albero di sintassi astratto, per effettuare un controllo sugli effetti che ogni istruzione presente nel programma produce sulle variabili ed inoltre si controlla che la sequenza di queste operazioni sia consistente
  3. **CodeGen** : Questo metodo si occupa, dopo che le fasi precedenti hanno verificato la correttezza della semantica dell'operazione, di calcolare una sequenza di istruzioni *bytecode* che corrispondano al costrutto descritto dal nodo corrente

- **Main** : Questo pacchetto contiene le uniche due classi contenenti delle funzioni main :

  1. **Compiler** : Questa classe si occupa di invocare il Parser *ANTLR* sul File in input. Ottenuto un Parse-Tree, costruisce un albero di sintassi astratta. Su questo invoca due discese ricorsive che controllano la correttezza semantica del programma (rispettivamente **CheckSemantics(AST)** e **CheckEffects(AST)**). Infine, se le fasi precedenti vanno a buon fine, si occupa di invocare **CodeGen(AST)** per ottenere in output delle istruzioni bytecode, che va a scrivere su file (**output/out.simple**)
  2. **VM** : Questa classe si occupa di definire il comportamento di ogni istruzione bytecode ed inoltre simula una virtual machine. Per tanto deve inizializzare ogni registro virtuale ed aggiornarli dopo ogni istruzione

- **Parser** : Questo pacchetto contiene tutte le classi che vengono generate automaticamente da ANTLR, oltre che a contenere la grammatica stessa(**SimplePlus.g4** ) :

  1. **Parser** : Il parser si occupa di chiamare il lexer il quale restituisce una lista di token che il parser va a strutturare in un Parse Tree

2. **Lexer** : Il lexer si occupa di ricevere in ingresso una lista di carattere e di fornire come output dei Token solo quando l'input e' lessicalmente coretto. Il lexer assume il comportamente di un automa a stati finiti con input i vari caratteri del file sorgente
3. **Visitor** : Visitor e' un intefaccia che l'utente deve implementare. Questa classe viene invocata dal Parser con input il Parse Tree e si occupa di generare l'albero di sintassi astratta andando ad eliminare qui componenti del albero che non sono necessari all'analisi semantica, ma hanno esaurito il loro scopo dopo l'analisi sintattica gia effetuata dal Parser

- **Util** : In questo paccheto sono presenti le classi di utilita' necessarie alle diverse fasi della compilazione.In particolare abbiamo :

1. **EnvironmentTypes** : Questa classe definisce l'ambiente utile all'analisi della corettezza dei tipi e dell'esistenza delle variabili. Per permettere questa analisi al suo interno la classe contiene una Linked List di HashMap. Ognuna di queste Hash Map definisce un ambito di visibilita' delle variabili. Ogni variabile che viene dichiarata e' salvata nell' HashMap che rappresenta lo scope attualmente piu' interno. In particolare viene inserito nel HashMap come chiave una stringa corrispondente al nome della variabile e come valore un'istanza della classe STEntryTypes. Inoltre ogni qual volta nel codice sorgente viene aperto un nuovo scope, una nuova HashMap viene appessa alla LinkedList ed ogni volta che uno scope viene chiuso l'ultima HashMap appesa viene rimossa dalla LinkedList.
2. **STEntryTypes** : Questa classe e' l'oggetto che usiamo come valore dell'HashMap contenuta nella classe EnvironmentTypes. Ogni istanza dell'oggetto tiene traccia del tipo della varibile corrispondente e del suo nesting level(ovvero il numero degli scope aperti al momento della dichiarazione della variabile)
3. **EnvironmentEffects** : Questa classe definisce l'ambiente utile all'analisi della correttezza della sequenza di operazioni che sono state effettuate sulla variabile. Anche in questo caso la classe contiene una LinkedList di HashMap per tenere traccia delle variabili e degli scope, con la differenza che in questo caso viene associata ad ogni varibile un'istanza della classe STEntryEffects ed ad ogni funzione un'instanza della classe STEntryEffectsFun
4. **STEntryEffects** : Questa classe e' l'oggetto che usiamo come valore dell'HashMap contenuta nella classe EnvironmentEffects. Ogni istanza dell'oggetto tiene traccia del tipo della variabile e del suo stato(che puo' assumere uno dei seguenti valori : $BOTTOM,RW,DELETE,TOP$)
5. **STEntryEffectsFun** : Questa classe e' l'oggetto che usiamo come valore dell'HashMap contenuta nella classe EnvironmentEffects in

corrispondenza della dichiarazione di funzioni. Ogni istanza dell'oggetto tiene traccia dello stato di ognuno dei parametri della funzione, ed inoltre se il parametro e' un riferimento ad una variabile dichiarata in un altro scope o meno

6. **EnvironmentCodeGen** : Questa classe definisce l'ambiente utile alla generazione del bytecode. Anche in questo caso la classe contiene una LinkedList di HashMap per tenere traccia delle variabili dichiarate e del loro scope di appartenenza. Ogni entry dell'HashMap e' in questo caso associata ad un'istanza della classe STEntryCodeGen. Inoltre questo ambiente contiene anche un metodo per il dispatch delle etichette che vengono utilizzate a livello di linguaggio bytecode per effettuare salti condizionati e salti regolari
7. **STEntryCodeGen** : Questa classe e' l'oggetto che usiamo come valore dell'HashMap contenuta nella classe EnvironmentCodeGen. Ogni istanza tiene traccia dell'offset (ovvero la cardinalita' con la quale la variabile è stata dichiarata all'interno dello scope corrispondente)e del nesting level (vedi StEntryTypes)

# 3 Sviluppo e scelte implementative

## 3.1 Analisi Lessicale e Sintattica

Per quanto rigurda l'analisi lessicale e sintattica sottolineaiamo che la grammatica non e' stata modificata rispetto a quanto descritto delle specifiche. Inoltre poichè il Parser ed il Lexer vengono generati automaticamente da ANTLR è stato solo necessario implementare l'intefaccia del Visitor. L'implementazione di questa interfaccia risulta essere molto banale per quanto visto a lezione, pertanto viene tralasciata la descrizione dettagliata.

Si riporta di seguito la grammatica del linguaggio per come definita nelle specifiche di progetto :

```
grammar SimplePlus;

// THIS IS THE PARSER INPUT

block       : '{' statement* '}';

statement   : declaration
            | assignment ';'
            | deletion ';'
            | print ';'
            | ret ';'
            | ite
            | call ';'
            | block;

declaration : decFun
            | decVar ;

decFun      : type ID '(' (arg (',' arg)*)? ')' block ;

decVar      : type ID ('=' exp)? ';' ;

type        : 'int'
            | 'bool'
            | 'void';

arg         : type ref? ID;

ref         : 'var';

assignment  : ID '=' exp;

deletion    : 'delete' ID;

print       : 'print' exp;
```

```
ret          : 'return' (exp)?;

ite          : 'if' '(' exp ')' statement ('else' statement)?
    ;

call         : ID '(' (exp(',' exp)*)? ')';

exp          : '(' exp ')'
             #baseExp
             | '-' exp
                       #negExp
             | '!' exp
                                                          #
               notExp
             | left=exp op=('*' | '/')               right=
               exp   #binExp
             | left=exp op=('+' | '-')               right=
               exp   #binExp
             | left=exp op=('<' | '<=' | '>' | '>=') right=
               exp   #binExp
             | left=exp op=('=='| '!=')              right=
               exp   #binExp
             | left=exp op='&&'                      right=
               exp   #binExp
             | left=exp op='||'                      right=
               exp   #binExp
             | call

               #callExp
             | BOOL

               #boolExp
             | ID
                              #varExp
             | NUMBER
                       #valExp;

// THIS IS THE LEXER INPUT

//Booleans
BOOL         : 'true'|'false';

//IDs
fragment
CHAR         : 'a'..'z' |'A'..'Z' ;
ID           : CHAR (CHAR | DIGIT)* ;

//Numbers
fragment
```

```
69 DIGIT        : '0'..'9';
   NUMBER       : DIGIT+;
71
   //ESCAPE SEQUENCES
73 WS               : (' '|'\t'|'\n'|'\r')-> skip;
   LINECOMMENTS     : '//' (~('\n'|'\r'))* -> skip;
75 BLOCKCOMMENTS    : '/*'( ~('/'|'*')|'/'~'*'|'*'~'/'
       |BLOCKCOMMENTS)* '*/' -> skip;
```

## 3.2 Analisi dei Tipi e gli Scope

L'analisi semantica, come detto viene implementata attraverso la visita dell'albero di sintassi astratta. La scansione dell'albero parte invocando l'analisi del nodo radice. In seguito il nodo radice, cosi' come poi fanno i nodi sottostanti, effettua l'analisi parziale o totale per quanto lo riguarda per poi invocare l'analisi sul nodo figlio (uno o piu' di uno) senza però interessarsi del tipo del nodo. Per invocare il metodo corretto, tutti i nodi implementano un interfaccia comune che descrive il comportamento generico. È poi ovviamente il *Dynamic dispatcher* che si occupa di invocare il metodo della sottoclasse corrispondente al tipo del nodo figlio.
Con questa premessa, possiamo dire che l'implementazione del metodo **CheckSemantics** per ogni classe è abbastanza coerente con quanto visto a lezione e spigato brevemente nelle sezioni precedenti. In questa fase ci sono solo due punti che si discostano dal metodo visto durante il corso e che quindi rappresentano delle scelte implementative prese dal gruppo :

- Durante l'analisi del nodo **SPReturn** che descrive il costrutto *return*, il quale forza l'uscita da una chiamata di funzione e restituisce il risultato della funzione, era neccario controllare che il tipo della variabile restituito corrispondesse al tipo di ritorno riportato al momento della dichiarazione della funzione. Questo si rende necessario poichè all'interno di una funzione, non solo è possibile utilizzare i parametri formali della funzione, ma anche variabili dichiarate all'interno del *Body* della funzione e variabili globali dichiarate prima della dichiarazione della funzione.
  Per ovviare a questo problema si è deciso di inserire nella classe *EnvironmentTypes* una LinkedList chiamata **Return_type_stack (RTS)**. Questa stuttura viene utilizzata per memorizzare il tipo di ritorno ogni volta che si inizia ad analizzare un nodo contenente la dichiarazione di una funzione. Inoltre prima di far partire l'analisi alla radice dell'*AST*, viene aggiunto alla stuttura *RTS* il valore *void* per tenere conto del tipo di ritorno della funzione *main* che si assume essere appunto *void*.

- Durante l'analisi del nodo **SPIfelse** che descrive il costrutto $if\{...\}else\{...\}$ era necessario necessario differenziare il comportamento sui due rami alternativi del costrutto.
  Prendiamo in considerazione per esempio il seguente frammento di codice :

```
1   int x=0;
    bool verify = someCondition();
3   if(verify){
        x+=1;
5       delete x;
    }
7   else{
        x=0;
9   }
```

In questo caso i due rami del costrutto sono entrambi corretti, ma poiche' nel ramo *then* e' presente il costrutto *delete*, che permette il riutilizzo dell'identificativo $x$ ma soltanto eliminando dall'ambiente tale nome, non è possibile analizzare il ramo *else* utilizzando l'ambiente *EnvironemntType* per come ottenuto alla fine dell'analisi del ramo precedente. Questo perche' l'identificativo $x$ utilizzato nel ramo else non verrebbe riconosciuto. L'analisi dei due rami quindi dovrebbe avvenire in "parallelo".
Per questo motivo si e' deciso di implementare un metodo *clone()* per la classe *EnvironmentTypes*. È inutile esplicitare le motivazioni per le quali questo metodo deve effettuare una copia profonda dell'oggetto. Grazie a questo metodo e' possibile effettuare un analisi corretta, andando a creare due copie dell'ambiente prima di analizzare i due rami del costrutto *if*{...}*else*{...} (ma ovviamente dopo aver analizzato la guardia ), per poi utilizzare le due copie rispettivamente per il primo e per il secondo ramo. In questo modo è possibile conciliare l'utlizzo di una variabile in un ramo e la sua eliminazione nell'altro.
NOTA : L'analisi degli effetti non avviene in questa fase per cui dopo l'analisi del costrutto, i due cloni non vengono uniti

## 3.3 Analisi degli Effetti

Anche la fase di Analisi degli Effetti si basa sulla visita dell'albero di sintassi astratta e l'invocazione dinamica del metodo **CheckEffects**. Proprio come l'analisi dei tipi, anche in questo caso viene utilizzata una classe ambiente per memorizzare lo stato del programma. Questa classe e' **EnvironmentEffects** di cui abbiamo già parlato nelle sezioni precedenti. Questa classe contiene oltre che i campi gia' descritti, anche dei metodi statici che descrivono gli operatori tra effetti visti a lezione. Grazie a questi metodi si e' proceduto ad implementare i meccanismi visti durante il corso. Le scelte implementative meritevoli di essere menzionate in questo caso sono le seguenti :

- Nell'analisi del nodo *SPDecVar* corrispondente al costrutto per la dichiarazione di una variabile e' necessario tenere in considerazione il fatto che una variabile con un nome precedentemente utilizzato puo' essere dichiarata nuovamente, solo a patto pero' che questa sia stata cancellata tramite

il costrutto *delete* e che la variabile non venga ridichiarata con un tipo diverso da quello precedente. Per permettere questi controlli sono state necessarie diverse modifiche. Per prima cosa l'ambiente *EnvironmentEffects* deve tenere conto anche del tipo delle variabili. Inoltre, nel controllare una dichiarazione di variabile a questo punto bisogna controllare che il nome non sia mai stato utilizzato, oppure che a quel nome corrisponda lo stato di variabile *DELETE* oltre che a controllare ovviamente che il tipo della nuova dichiarazione e quello della dichiarazione precedente coincidano

- Come visto a lezione per poter analizzare la correttezza a livello di *Effetti*, quando si parla di funzioni ricorsive, e' necessario calcolare l'effetto della funzione tramite un calcolo del punto fisso. Tutto cio' viene implementato nella classe **SPDecFun**. Per effettuare questo calcolo e' necessario inizializzare lo stato dei parametri formali al valore $BOTTOM$, ovvero assumendo inizialmente che la funzione sia $BOTTOM \rightarrow BOTTOM$ per ogni parametro. Alla fine di ogni iterazione, il corpo della funzione avra' modificato lo stato dei parametri. A questo punto lo stato finale indichera' in che modo il corpo della funzione ha avuto effetto sui parametri, per tanto si aggiorna l'effetto della funzione, il quale verra' utilizzato alla prossima iterazione per ripetere il calcolo tenendo in considerazione la chiamata ricorsiva. Quando una delle iterazioni non indica un comportamento differente da quello fin li calcolato per la funzione, allora si e' raggiunto il punto fisso.
  Detto questo, notiamo che gli ambienti che utilizziamo per fare questo calcolo non hanno nulla a che vedere con l'ambiente esterno alla dichiarazione della funzione. Per non apportare nessuna modifica all'ambiente fin li ottenuto e quindi allo stato di variabili che non hanno a che vedere con la funzione, ma anche per non tenere in considerazione gli effetti calcolati durante ognuna delle iterazioni, e' stato implementato un metodo *clone()* per la classe *EnvironmentEffects* cosicche' sia possibile utilizzare un ambiente opportuno ad ogni nuova iterazione del calcolo del punto fisso.
  NOTA : Questo metodo *clone()* e' stato utilizzato anche per implementare il meccanismo visto a lezione per il costrutto di scelta condizionale *if* (un clone per ogni ramo dell'*if* e poi si uniscono i due cloni tramite l'operatore max). La descrizione qui non e' dettagliata poiche' l'implementazione e' totalmente simile a quella vista durante il corso

```
1   int y=0;
    void f(int someparam){
3       /* Qui voglio conoscere l'ambiente per sapere
         * se l'ID 'y' esiste, ma non voglio
5        * modificare il vero ambiente perche' a
         * questo punto della compilazione sto
7        * processando soltanto una dichiarazione
         */
9       delete y;
    }                          [...] f(4);
```

## 3.4 Code Generation e Interprete

Anche le fase di generazione del bytecode avviene attraverso la visita ricorsiva dell'albero di sintassi astratta. In questo caso il compilatore utilizza la classe **EnvironmentCodeGen** per memorizzare le informazioni riguardanti le variabili. Prima di trattare i dettagli di implementazione rilevanti e' necessario un preambolo riguardante due aspetti :

1. La struttura del ***Frame***: Il *Frame* e' una struttura dati composta da celle consecutive contenenti tutte le informazioni relative ad una chiamata di funzione (prima delle quali l'invocazione della funzione main ). In un dato punto del programma nella cpu (virtuale) e' presente un registro denominato **$FP (Frame Pointer)** che contiene il primo indirizzo del *Frame* corrente, ovvero l'indirizzo dal quale iniziano ad essere scritte le informazioni riguardanti la chiamata di funzione corrente.
   A questo punto e' necessario definire una struttura costante per il *Frame* in modo da capire cosa rappresenta ogni cella a partire dall'indirizzo puntato da *$FP*. Al primo indirizzo (*0($FP)*) e' presente il contenuto precedente del registro *$FP*, ovvero il puntatore al *Frame* del chiamante rispetto alla chiamata di funzione corrente. Al secondo indirizzo (*-Dim_parola($FP)*) e' presente il contenuto precedente del registro *$RA*. Bisogna dire infatti che ogni istruzione ha un proprio identificativo. Il registro *$RA* contiene ad ogni istante dell'esecuzione, l'identificativo dell'istruzione dalla quale il chiamante deve riprendere alla fine dell'esecuzione della funzione corrente. Al terzo indirizzo (*-Dim_parola*2($FP)*) e' presente il nesting level del *Frame* corrente, ovvero il livello di ricorsione che ha la chiamata di funzione corrente. Parleremo meglio di questo valore in seguito. Dal quarto indirizzo in poi vengono memorizzate tutti parametri formali e le variabili dichiarate all'interno dello scope della funzione.

2. Design dell'***Interprete*** e dell'***Instruction set*** : L'interprete del linguaggio e' una macchina a pila, che lavora su registri nella cpu (virtuale). Le operazioni infatti non prendono gli operandi direttamente dalla memoria centrale, ma hanno bisogno di caricare i dati su registri quali ad esempio *$a0* e *$t0*. Oltre a questi registri abbiamo i gia' citati *$fp* e *$ra* ed il registro *$ip (Instruction pointer)* che punta alla prossima istruzione da eseguire e viene quindi letto in fase di *fetch*. Inoltre sono presenti i registri *$al (Access Link)* e *$sp (Stack Pointer)*. Questi servono rispettivamente a risalire i *Frame* per andare ad accedere alle variabili non locali ed ad accedere alla memoria centrale in lettura o scrittura (*$sp* punta infatti alla prossima cella disponibile per la scrittura).
   La *Macchina Virtuale* dispone di 10000 (diecimila) celle che compongono la memoria centrale (virtuale) ed ogni parola occupa 4 di queste celle.
   Si e' scelto di non inserire tra le istruzioni riconosciute dall'*Interprete* le macro viste a lezione : *Push* e *Pull*. Queste vengono sostituite dalla combinazione delle istruzioni **ADDI** e **LW/SW** (rispettivamente). Tra le

altre istruzioni troviamo : **LI**(Per caricare una costante in un registro) **ADD**(Per sommare il valore di due registri in memoria), **SUBI**(Per sottrarre il valore di un registro ed una costante), **MOVE**(Per muovere il valore di un registro in un altro), **JR**(Legge il valore riportato nel registro e salta a quell'istruzione), **JAL**(Salta ad un etichetta, salvando prima in $ra il valore del registro $ip), **B**(Salto incondizionato ad un etichetta), **BEQ**(Salto condizionato ad etichetta se i due argomenti sono uguali), **BGE**(Salto condizionato ad etichetta se il secondo argomento e' maggiore o uguale al primo), **BLE**(Salto condizionato ad etichetta se il secondo argomento e' minore o uguale al primo), **PRINT**(Stampa il valore del registro), **NEG**(Nega il valore del registro), **MULT**(Moltiplica i valori nei registri), **DIV**(Dividi i valori nei registri), **HALT**(Interrompi l'esecuzione del programma senza errori).
Si sottolinea il fatto che si e' preferito utlizzare un *Instruction Set* al livello piu' basso possibile, senza inserire macro o istruzioni troppo complesse.

Con questa lunga premessa possiamo andare ad analizzare l'unica scelta implementativa effettuata in questa fase (che e' il motivo per cui, come detto, si e' deciso di aggiungere un branch al repository, dove poter sviluppare questa feature).
In particolare, il linguaggio permette l'utilizzo di variabili globali all'interno di funzioni. Per implementare questa caratteristica a livello di linguaggio macchina si sarebbe potuto aggiungere alla funzione dei parametri formali che sostituissero la variabile globale. Si e' scelto di non percorrere questa via, poiche' questo avrebbe richiesto un ulteriore pre-scansione della funzione per analizzare le variabili globali nominate, per poi modificare il comportamento della funzione stessa affinche' si riferisse al parametro aggiuntivo laddove prima si riferiva alla variabile globale.
Piuttosto che fare quanto detto si e' scelto di implementare un modo per percorrere a ritroso i vari *Frame* per accedere a quello contenente la variabile globale corretta. Per permettere questo accesso dinamico si rende necessario conoscere dinamicamente il *Nesting Level* corrente in ogni momento dell'esecuzione e confrontarlo con quello relativo alla variabile, noto a tempo di compilazione. Per conoscere questo *Nesting Level*, come descritto nelle sezioni precedenti, questo viene salvato come terzo parametro di ogni *Frame* nel momento dell' inizializzazione di quest'ultimo. Quando il programma deve accedere ad una variabile, a livello di codice macchina non si fa altro che salire di *Frame* (tramite meccanismo di Access Link ed Old Frame Pointer) e decrementare il valore di *Nesting Level* (che verra' salvato in un registro) fin tanto che questo non raggiunge il valore corrispondente a quello della variabile da accedere.

# 4 Scaricare ed inizializzare il software

## 4.1 Introduzione

Durante la fase di sviluppo, è stato utilizzato il software per il versioning del codice sorgente **GIT** in combinazione dal lato server con il servizio **GITHUB**. Sul repository ci sono due branch. In particolare sul branch **runtime-nl** sono state implementate delle feature aggiuntive alla versione presente sul branch **master**, pertanto la prima è da ritenersi la versione del progetto completa.

## 4.2 Download ed installazione

Per eseguire il codice, per prima cosa e' necessario scaricare il sorgente dalla repository :

```
1 git clone "https://github.com/AlecioP/CompilerAntlr"
```

Entrare nella directory appena creata :

```
1 cd CompilerAntlr
```

Accedere al branch con le feature piu' aggiornate :

```
1 git checkout runtime-nl
```

Infine per avere informazioni dettagliate sul funzionamento del software :

```
1 make help
```