|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Documento di manutenzione   |  |  | | --- | --- | | Riferimento |  | | Versione | 1.2 | | Data | 06/10/2019 | |

RevisionHistory

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 30/09/2019 | 1.0 | Prima Stesura | F. Mazzotta |
| 03/10/2019 | 1.1 | Aggiunta paragrafo progettazione ed impact analysis | F. Mazzotta |
| 06/10/2019 | 1.2 | Inserimento diagrammi | F. Mazzotta |

Sommario

[RevisionHistory 2](#_Toc21360151)

[1. Il compilatore Yaspl 2.0 4](#_Toc21360152)

[1.1 Analisi lessicale. 4](#_Toc21360153)

[1.2 Analisi sintattica. 5](#_Toc21360154)

[1.3 Analisi semantica. 10](#_Toc21360156)

[1.4 Generazione del codice. 12](#_Toc21360157)

[2. Progettazione della modifica 12](#_Toc21360158)

[3 Impact Analisys 13](#_Toc21360159)

[4. Ulteriori correzioni 14](#_Toc21360161)

1. Il compilatore Yaspl 2.0

Il progetto software che si vuole sottoporre a manutenzione è il compilatore YASPL 2.0. Il compilatore traduce il linguaggio YASPL 2.0 in linguaggio C. Per fare ciò il codice YASPL 2.0 è stato sottoposto ad analisi lessicale, sintattica e semantica. Dopo queste analisi viene generato il corrispondente codice C. L’ implementazione di queste analisi viene descritto di seguito.

1.1 Analisi lessicale.

L’analisi lessicale è stata realizzata con il tool JFlex; al suo interno sono state definite le espressioni regolari che riconoscono i token ed è stato inserito il codice che inserisce il token nella symbol table. Il tool crea la componente **Lexer** che implementa l’interfaccia **sym.**

L’analizzatore lessicale implementato riconosce i seguenti token:

*HEAD* ***head***

*START* ***start***

*SEMI* ***;***

*INT* ***int***

*BOOL* ***bool***

*DOUBLE* ***double***

*NAME* [a-zA-Z0-9\_]\*

*COMMA* ***,***

*DEF* ***def***

*LPAR* ***(***

*RPAR* ***)***

*COLON* ***:***

*LGPAR* ***{***

*RGPAR* ***}***

*READ* ***<-***

*WRITE* ***->***

*PLUS* ***+***

*MINUS* ***-***

*TIMES* ***\****

*DIV* ***/***

*INT\_CONST* [0-9]\*

*DOUBLE\_CONST* [0-9.]\*

*STRING\_CONST pattern per stringhe*

*TRUE* ***true***

*FALSE* ***false***

*ASSIGN* ***=***

*IF* ***if***

*THEN* ***then***

*WHILE* ***while***

*DO* ***do***

*ELSE* ***else***

*GT* ***>***

*GE* ***>=***

*LT* ***<***

*LE* ***<=***

*EQ* ***==***

*NOT* ***not***

*AND* ***&&***

*OR* ***||***

*UMINUS* ***-***

*COMMENTIINLINEA* **//**

*COMMENTISUPIURIGA* ***/\* \*/***

*CONTENUTOCOMMENTO*

*TAB*

*NEWLINE*

*WHITESPACE*

1.2 Analisi sintattica.

L’analisi sintattica è stata prodotta con il tool ***JavaCUP*** che genera un parser LALR(1). La grammatica del linguaggio è la seguente:

***Programma → HEAD Decls START Statements***

***Decls → Var\_decl Decls***

***| Def\_decl Decls***

***| ε***

***Statements → Stat Statements***

***| Stat***

***Var\_decl → Type Vars SEMI***

***Type → INT | BOOL | DOUBLE***

***Vars → NAME COMMA Vars***

***| NAME***

***Types → Type COMMA Types***

***| Type***

***Def\_decl → DEF NAME LPAR Var\_decls RPAR COLON Par\_decls Body***

***Var\_decls → Var\_decl Var\_decls***

***| ε***

***Par\_decls → Var\_decl Par\_decls***

***| Var\_decl***

***Body → LGPAR Var\_decls Statements RGPAR SEMI***

***Comp\_stat→ LGPAR Statements RGPAR***

***Stat → Comp\_stat***

***| Simple\_stat***

***Simple\_stat→ Vars READ Types***

***| Out\_values WRITE SEMI***

***| NAME ASSIGN Expr SEMI***

***| NAME LPAR Exprs COLON Vars RPAR SEMI***

***| IF LPAR Bool\_expr RPAR THEN Comp\_stat ELSE Comp\_stat SEMI***

***| IF LPAR Bool\_expr RPAR THEN Comp\_stat SEMI***

***| WHILE LPAR Bool\_expr RPAR DO Comp\_stat SEMI***

***Out\_values → Expr COMMA Out\_values***

***| STRING\_CONST COMMA Out\_values***

***| Expr***

***| STRING\_CONST***

***Exprs → Expr COMMA Exprs***

***| Expr***

***Expr → INT\_CONST***

***| DOUBLE\_CONST***

***| Expr Arith\_op Expr***

***| NAME***

***| UMINUS Expr***

***| LPAR Expr RPAR***

***Arith\_op → PLUS | MINUS | TIMES | DIV***

***Bool\_expr → TRUE***

***| FALSE***

***| Bool\_expr Bool\_op Bool\_expr***

***| NOT Bool\_expr***

***| Expr Rel\_op Expr***

***| LPAR Bool\_Expr RPAR***

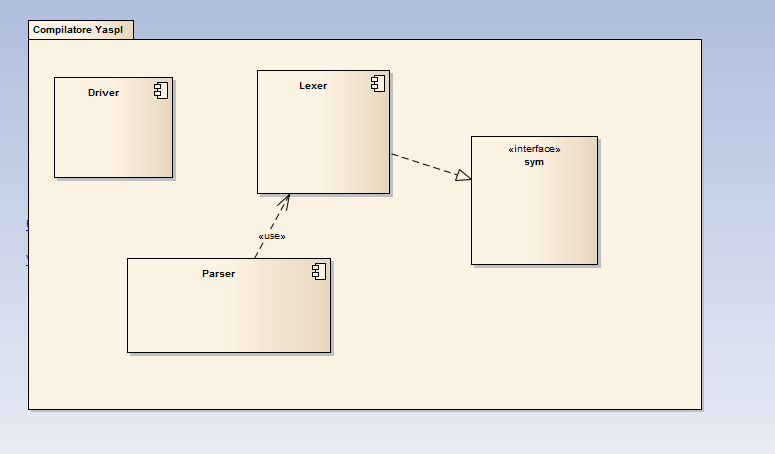
***Bool\_op → AND | OR***

***Rel\_op → GT | GE | LT| LE | EQ***

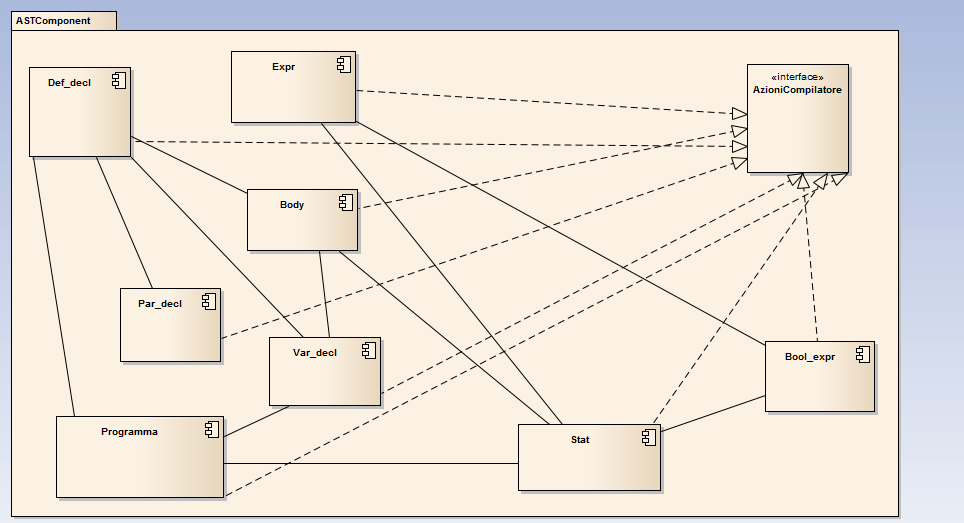
Per realizzare le varie parti che formano l’abstract syntax tree sono state realizzate le seguenti componenti:

* **Programma**: è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce a tutto il programma***.*** Al suo interno sono presenti le classi interne ***ProgramOp*** e ***Decls*** che sono sottoclassi della classe Programma e quindi implementano i metodi definiti nell’interfaccia AzioniCompilatore. Queste sottoclassi rappresentano le possibili forme che può assumere il non terminale di tipo Programma. Sono stati inseriti due metodi statici che permettono di istanziare gli oggetti delle classi interne.
* **Stat:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce alle varie istruzioni del programma. Al suo interno sono presenti le classi interne ***AssignOp,*** che corrisponde all’istruzione di assegnazione; ***ReadOp,*** istruzione che legge i dati in ingresso; ***WriteOp,*** istruzione che stampa i dati in output; ***WhileOp,*** corrisponde all’istruzione while; ***IfThenOp,*** corrisponde all’istruzione if senza lato else***; CompStatOp,*** corrisponde ad un blocco di istruzioni appartenenti al lato vero di un while o if; ***CallOp,*** corrisponde all’istruzione per chiamare una funzione; ***IfThenElseOp.*** corrisponde all’istruzione if con lato else. Sono tutte sottoclassi della classe Stat e quindi implementano i metodi definiti nell’interfaccia AzioniCompilatore; rappresentano le possibili forme che può assumere il non terminale di tipo Stat. Sono stati implementati dei metodi statici per poter istanziare gli oggetti delle classi interne.
* **Def\_decl:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce alle funzioni definite all’interno del programma. Al suo interno è presente la classe interna ***ProcDeclOp*** che è una sottoclasse della classe Def\_decl e quindi implementa i metodi definiti nell’interfaccia AzioniCompilatore. Questa sottoclasse rappresenta la possibile forma che può assumere il non terminale di tipo Def\_decl. È stato implementato un metodo statico per poter istanziare l’oggetto della classe interna.
* **Par\_decl:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce agli output restituiti da una funzione. Al suo interno è presente la classe interna ***ParDeclOp*** che è una sottoclasse della classe Par\_decl e quindi implementa i metodi definiti nell’interfaccia AzioniCompilatore. Questa sottoclasse rappresenta la possibile forma che può assumere il non terminale di tipo Par\_decl. È stato implementato un metodo statico per poter istanziare l’oggetto della classe interna.
* **Body:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce al corpo che contiene le istruzioni di una funzione. Al suo interno è presente la classe interna: ***BodyOp*** che è una sottoclasse della classe Body e quindi implementa i metodi definiti nell’interfaccia AzioniCompilatore. Questa sottoclasse rappresenta la possibile forma che può assumere il non terminale di tipo Body. È stato implementato un metodo statico per poter istanziare l’oggetto della classe interna.
* **Expr:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce alle espressioni usate all’interno del programma. Al suo interno sono presenti le classi interne: ***Priority,*** che rappresenta espressioni con le parentesi; ***Type,*** che rappresenta i tipi di dati delle variabili; ***Binex,*** che rappresenta espressioni aritmetiche; ***Unex,*** che rappresenta espressioni unarie; ***StringConst,*** che rappresenta costanti di stringhe; ***Identfier,*** che rappresenta gli identificatori delle variabili; ***IntDoubleConst,*** che rappresenta costanti numeriche. Sono sottoclassi della classe Expr e quindi implementano i metodi definiti nell’interfaccia AzioniCompilatore e rappresentano le possibili forme che può assumere il non terminale di tipo Expr. Sono stati implementati i metodi statici per poter istanziare gli oggetti delle classi interne.
* **Bool\_expr:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce alle espressioni booleane all’interno del programma. Al suo interno sono presenti le classi interne: ***Priority2,*** che rappresenta espressioni booleane con le parentesi; ***BoolOp,*** che rappresenta espressioni con operatori booleani; ***NotUnex,*** che rappresenta espressioni booleane unarie; ***RelOp,*** che rappresenta espressioni booleane relazionali; ***IdentfierBool,*** che rappresenta identificatori di espressioni booleane; ***BoolConst,*** che rappresenta valori di costanti booleane. Sono sottoclassi della classe Bool\_expr e quindi implementano i metodi definiti nell’interfaccia AzioniCompilatore e rappresentano le possibili forme che può assumere il non terminale di tipo Bool\_expr. Sono stati implementati dei metodi statici per poter istanziare gli oggetti delle classi interne.
* **Var\_decl:** è una classe astratta che implementa i metodi dell’interfaccia ***AzioniCompilatore.*** Questa classe rappresenta il non terminale che si riferisce alla dichiarazione delle variabili. Al suo interno sono presenti le classi interne: ***VarDeclOp*** e ***VarOp*** che sono sottoclassi della classe Var\_Decl e quindi implementano i metodi definiti nell’interfaccia AzioniCompilatore. Queste sottoclassi rappresentano le possibili forme che può assumere il non terminale di tipo Var\_decl. Sono stati implementati dei metodi statici per poter istanziare gli oggetti delle classi interne.

All’interno del tool di JavaCup è stata definita la grammatica del linguaggio e le relative operazioni per costruire l’abstract syntax tree (AST). Sono stati dichiarati i terminali del linguaggio che corrispondono ai token riconosciuti dall’analisi lessicale, e i non terminali che corrispondono a variabili di tipo Stat, Def\_decl, Programma, Var\_decl, Par\_decl, Body, Expr ed Bool\_expr. L’AST viene generato in questo modo: per ogni produzione della grammatica viene chiamato il metodo statico della classe a cui corrisponde il tipo del non terminale; viene crea l’oggetto della classe interna che si riferisce alla produzione in questione; l’oggetto creato viene assegnato al riferimento della variabile del non terminale. Infine l’AST risultante è dato da tutti i collegamenti che mettono in relazione le varie componenti. L’output prodotto dal tool è la componente **Parser.** Per ottenere l’AST basta invocare il suo metodo ***parse().*** In questo modo otteniamo l’oggetto iniziale dell’AST, che sarebbe un oggetto di tipo Programma. Per verificarne la correttezza l’AST viene stampato su un file XML. Per realizzare la stampa ogni componente dell’AST implementa il metodo ***drawComponent()*** dell’interfaccia AzioniCompilatoreche stampa su XML la sua corrispondente parte dell’AST. Si rappresenta il diagramma delle componenti coinvolte nell’analisi sintattica.



**Diagramma relativo alle componenti generate da JavaCup e JFlex.**



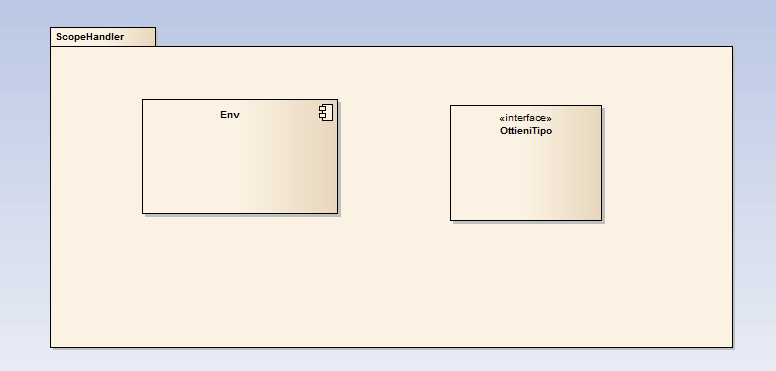
**Diagramma relativo alle componenti che formano l’abstract syntax tree**

1.3 Analisi semantica.

L’analisi semantica viene effettuata sull’AST prodotto durante l’analisi sintattica Per realizzare l’analisi semantica ogni componente dell’AST implementa il metodo ***startScoping()*** dell’interfaccia AzioniCompilatori a seconda della parte che occupa all’interno dell’AST. Infatti abbiamo che:

* La sottoclasse ***ProcDeclOp*** di Def\_decl fa controlli sull’esistenza di una funzione che ha già lo stesso nome, che non ci siano variabili con il nome della funzione, e che non ci sia più di un valore da restituire in output.
* La sottoclasse ***ParDeclOp*** di Par\_decl controlla che nella dichiarazione della variabile di output non ci sia più di un identificatore.
* La sottoclasse ***VarDeclOp*** di Var\_decl controlla che nella dichiarazione degli argomenti di input nella definizione della funzione non ci sia più di un identificatore e la presenza delle variabili all’interno di uno scope.
* La sottoclasse ***ReadOp*** di Stat effettua controlli sui tipi di dati delle variabili da acquisire e che gli argomenti della read devono essere di numero uguale.
* La sottoclasse ***AssignOp*** di Stat effettua controlli sull’esistenza della variabile a cui bisogna assegnare il valore e fa il controllo dei tipi di dati tra la variabile di assegnazione e sull’espressione di assegnazione.
* La sottoclasse ***CallOp*** di Stat effettua controlli sull’esistenza della funzione da chiamare, sulla quantità di argomenti passati alla funzione, in maniera tale che siano uguali al numero di parametri definiti dalla funzione; sui tipi di dati delle variabili da passare alla funzione, in maniera tale che siano dello stesso tipo di dati definiti dalla funzione, sui tipi di dati delle variabili a cui assegnare il valore restituito dalla funzione.
* Le sottoclassi ***IfThenOp***, ***IfThenElse***, ***WhileOp*** di Stat effettuano controlli sui tipi di dati delle variabili in maniera tale che la condizione sia di tipo booleano.
* La sottoclasse ***Binex*** di Expr effettua i controlli sui tipi di dati delle variabili coinvolti nell’espressione aritmetica.
* La sottoclasse ***Identifier*** di Expr effettua un controllo sull’esistenza della variabile all’interno di uno scope.
* La sottoclasse ***Unex*** di Expr effettua un controllo sul tipo di dato della variabile su cui viene applicato l’operatore.
* La sottoclasse ***BoolOp*** di Bool\_expr effettua un controllo sui tipi di dati delle variabili coinvolte nell’operazione
* La sottoclasse ***IdentifierBool*** di Bool\_expr effettua un controllo sull’esistenza della variabile all’interno di uno scope.
* La sottoclasse ***NotUnex*** di Bool\_expr effettua un controllo sul tipo di dato della variabile coinvolta nell’operazione
* La sottoclasse ***RelOp*** di Bool\_expr effettua un controllo sui tipi di dati delle variabili coinvolte nell’operazione

Per gestire lo scope delle variabili e i parametri delle funzioni le componenti dell’AST utilizzano la classe ***Env***. Le classi Par\_decl, Expr e Bool\_expr implementano il metodo ***getType()*** dell’interfaccia ***OttieniTipo.*** Questo metodo viene utilizzato per conoscere i tipi di dati delle varie espressioni del programma e il tipo di dato restituito dalla funzione.



**Diagramma delle componenti coinvolte nell’analisi semantica.**

1.4 Generazione del codice.

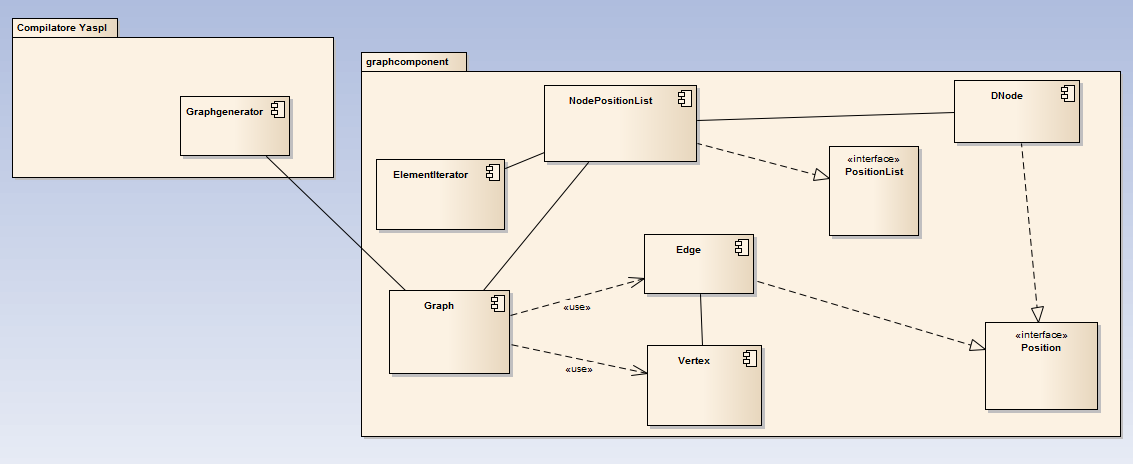
La generazione del codice avviene usando l’AST calcolato dopo l’analisi semantica. Ogni componente dell’AST implementa il metodo ***scriviCodice()*** dell’interfaccia AzioneCompilatore, con il quale scrive su un file la corrispondente istruzione C in base al ruolo che occupa nell’AST. Per avviare il processo di compilazione bisogna eseguire il metodo ***main()*** all’interno della classe ***Driver.***

2. Progettazione della modifica

Si vuole modificare il compilatore inserendo un ulteriore analisi del codice. Si vuole implementare l’analisi di data flow per rilevare eventuali anomalie che derivano dall’utilizzo delle variabili. Per implementare ciò sarà necessario costruire un control flow graph (CFG) ed una volta ottenuto il CFG su di esso verrà applicato l’algoritmo di reaching definition. Il CFG verrà costruito usando l’AST ottenuto dalle varie analisi precedenti. Una volta ottenuto, verranno memorizzate su un file le informazioni relative ad esso e tramite un apposito visualizzatore di grafi verrà stampato graficamente. L’algoritmo di reaching definition usa il CFG calcolato e verrà implementato nel seguente modo: prende in input il nome della variabile, verifica se esiste la variabile o no. Se esiste, inizia a scorrere tutti i nodi del grafo. Se trova un nodo in cui la variabile viene definita per la prima volta viene istanziato un token che prende in input il nome della variabile e l’identificativo del nodo, che può essere espresso nel seguente formato: ***<nodo,variabile>***. Una nuova definizione della variabile porta ad aggiornare il nodo del token della variabile su cui viene applicato l’algoritmo di reaching definition. Se trova un uso della variabile in un nodo viene salvata una tripla che indica che la definizione della variabile al nodo i-esimo (che corrisponde al nodo memorizzato nel token della variabile definita) raggiunge l’uso al nodo j-esimo appena esplorato. La tripla viene rappresentata nel seguente formato: ***<nodo Definizione, nodo Uso, variabile>***. Una volta terminato ciò, vengono usate le triple calcolate per inserire gli archi sul CFG che indicano i cammini trovati dall’algoritmo. I vari cammini delle varie definizioni della variabile vengono stampati con colori diversi. Le informazioni del nuovo CFG verranno memorizzate su un file e poi verrà stampato graficamente tramite l’apposito visualizzatore di grafi.

3 Impact Analisys

Per poter effettuare la modifica sarà necessario creare delle nuove componenti per poter rappresentare la struttura dati del grafo e le sue relative parti che lo costituiscono, creare la componente che permette di stampare ed eseguire l’algoritmo di reaching definition che esplora il grafo ottenuto. La costruzione del grafo comporterà l’aggiunta di un ulteriore metodo in tutte le componenti che formano l’AST. In questo modo ogni componente creerà il nodo in base a ciò che rappresenta e creerà un arco che unisce il nodo che ha appena creato e l’ultimo nodo creato. Quindi sarà necessario aggiungere il metodo ***buildControlFlow()*** nell’interfaccia AzioniCompilatori e di conseguenza questo metodo sarà implementato da tutte le componenti dell’AST. Saranno soggette a testing di regressione tutte le componenti dell’AST che eseguono l’analisi lessicale, sintattica e semantica, e la generazione del codice in quanto non risultano essere modificate. Di seguito il diagramma delle componenti aggiunte.



4. Ulteriori correzioni

Durante il processo di manutenzione sono stati corretti i seguenti bug:

* Riconoscimento dell’istruzione unaria per rappresentare i numeri negativi in quanto precedentemente non venivano riconosciute a causa delle precedenze sui segni non inseriti nel parser.
* Miglioramenti sulla gestione delle operazioni delle priorità delle parentesi in quanto venivano calcolate in modo scorretto
* Riparazione del riconoscimento semantico del nome delle funzioni, poiché era possibile inserire il nome delle funzioni e delle variabili con lo stesso identificativo.