Sommario

1 Modifiche grammatica

1.1 Modifica Ricorsioni

Nelle produzioni che creano delle liste, in particolare

ExprList ::= Expr COMMA ExprList | Expr
ElifList ::= Elif ElifList | /* vuoto */
StatList ::= Stat StatList | Stat
VarDeclList ::= VarDecl VarDeclList | /* vuoto */
ProcList ::= Proc ProcList | Proc
ResultTypeList ::= ResultType COMMA ResultTypeList | ResultType

notiamo la presenza di ricorsione destra. Questo è un problema nella creazione delle liste in quanto queste verranno create al contrario dato l'utilizzo di un parser bottom-up.
Il problema può essere risolto andando a modificare tali produzioni nella ricorsione e facendole diventare da ricorsive destre a ricorsive sinistre. Così facendo le produzioni saranno modificate in tal modo

ExprList ::= ExprList COMMA Expr | Expr
ElifList ::= ElifList Elif | /* vuoto */
StatList ::= StatList Stat | Stat
VarDeclList ::= VarDeclList VarDecl | /* vuoto */
ProcList ::= ProcList Proc | Proc
ResultTypeList ::= ResultTypeList COMMA ResultType | ResultType

e come risultato avremo le liste create nell'ordine corretto.
Esempio del perché non funziona:

StatList -> Stat StatList
StatList -> Stat
Stat -> if
Stat -> for
Stat -> while

Ricordiamo che operiamo su di un parser bottom-up: per vedere come funziona facciamo una derivazione destra e prendiamo i risultati al contrario

input: if for while

StatList <=(6) Stat StatList <=(5) Stat Stat StatList <=(4) Stat Stat Stat
<=(3) Stat Stat while <=(2) Stat for while <=(1) if for while

Gli statement vengono aggiunti nell'array quando riduciamo uno “Stat” o “Stat StatList” in “StatList”. Come notiamo nella derivazione, la produzione che crea la lista, essendo ricorsiva destra, aggiunge gli stati dall'ultimo al primo:
Stat Stat Stat (4)=> Stat Stat StatList (5)=> Stat StatList (6)=> StatList

con la ricorsione sinistra invece, cambiando la prima produzione in “StatList -> StatList Stat” avremo:

StatList <=(6) StatList Stat <=(5) StatList while <=(4) StatList Stat while <=(3)
StatList for while <=(2) Stat for while <=(1) if for while

in questo caso possiamo vedere che gli Stat vengono ridotti in StatList con l'ordine
dato in input.

1.2 Tipi di ritorno di una funzione

Grammatica modificata per fare in modo che possiamo dichiarare funzioni che ritornano solo void o una lista di tipi come int, float, bolean e string etc... Questa modifica ci permette di verificare nell’analisi sintattica la correttezza della firma della funzione, invece che fare i controlli nell’analisi semantica.
Nel caso in cui la funzione restituisce void, l'array dei tipi di ritorno è creato, ma sarà vuoto.
Nelle produzioni Proc sostituiamo il non terminale ResultTypeList con ResultTypeListWithVoid e modifichiamo la grammatica da

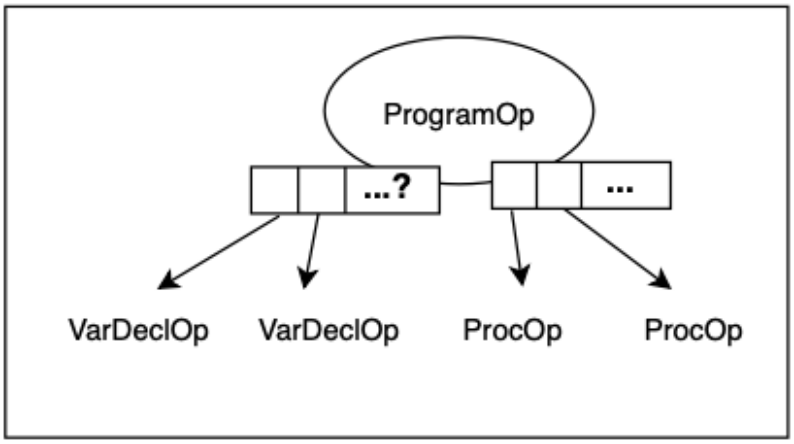
ResultTypeList ::= ResultType | ResultTypeList COMMA ResultType;
ResultType ::= Type | VOID;

a

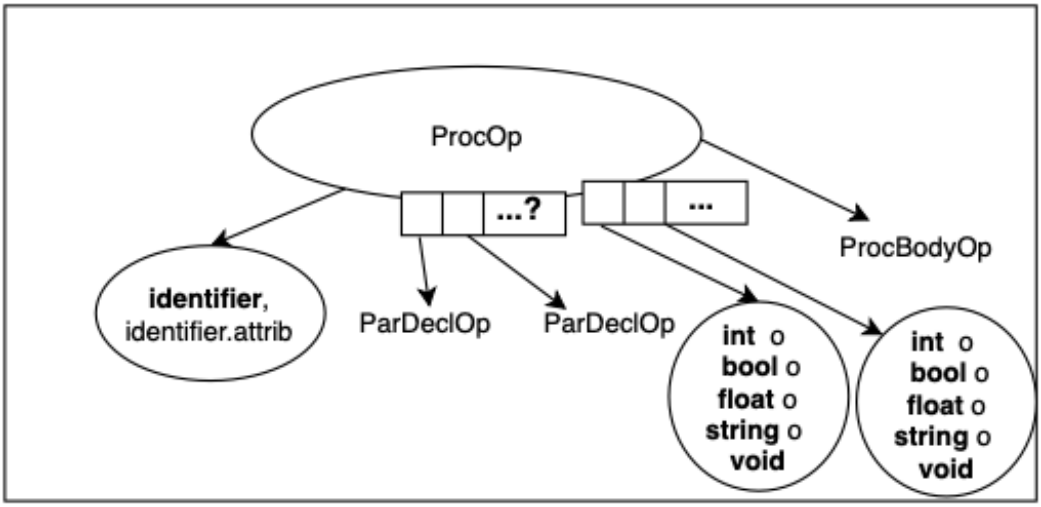
ResultTypeListWithVoid ::= ResultTypeList | VOID;
ResultTypeList ::= Type | ResultTypeList COMMA Type;

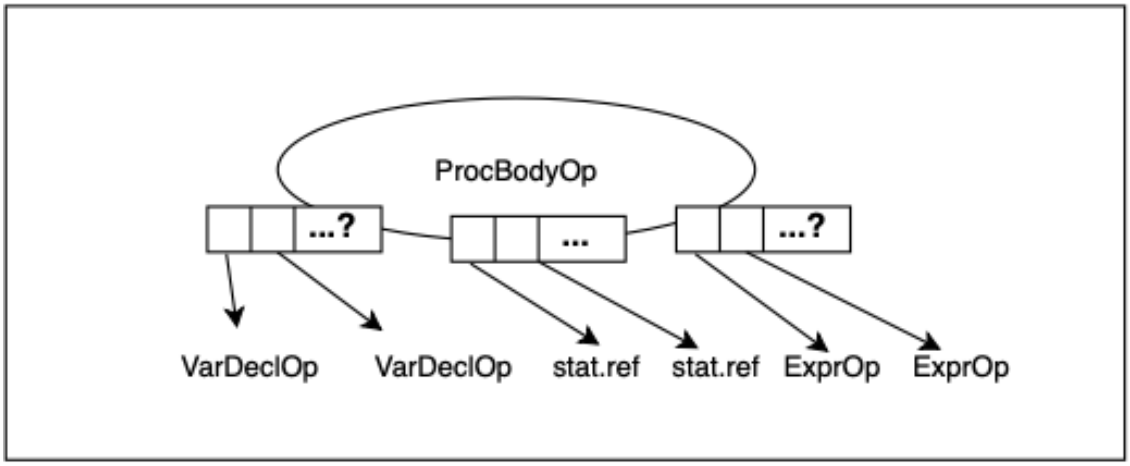
2 Diagrammi AST

Program

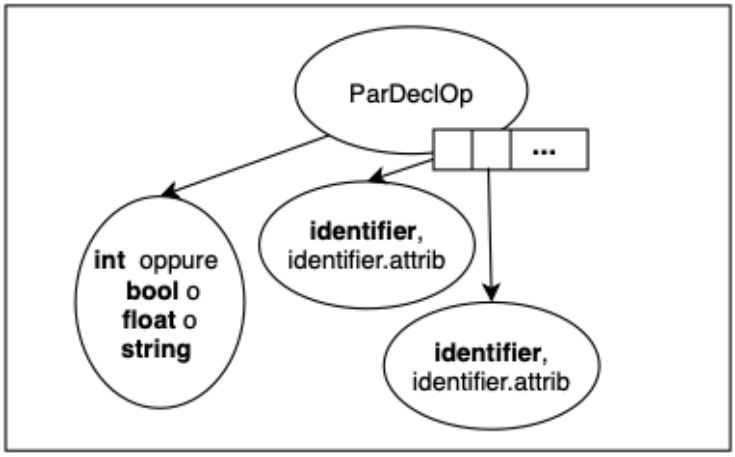


Proc

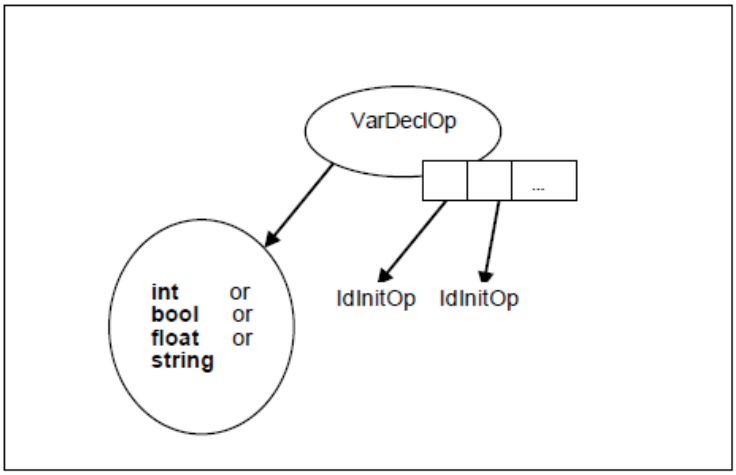


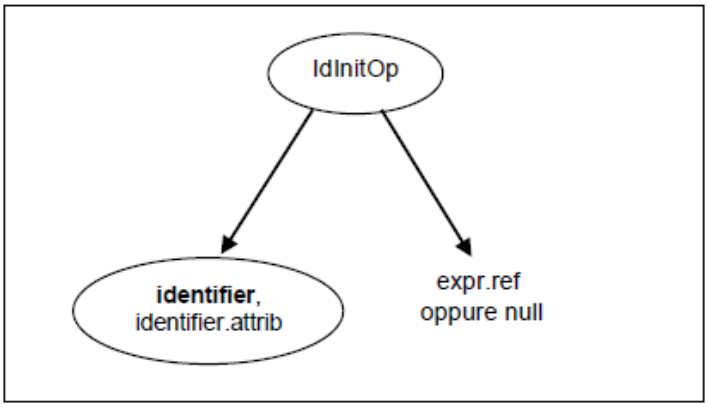


Par_decl

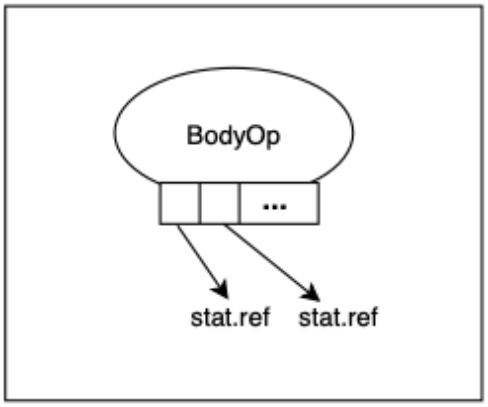


Var_decl

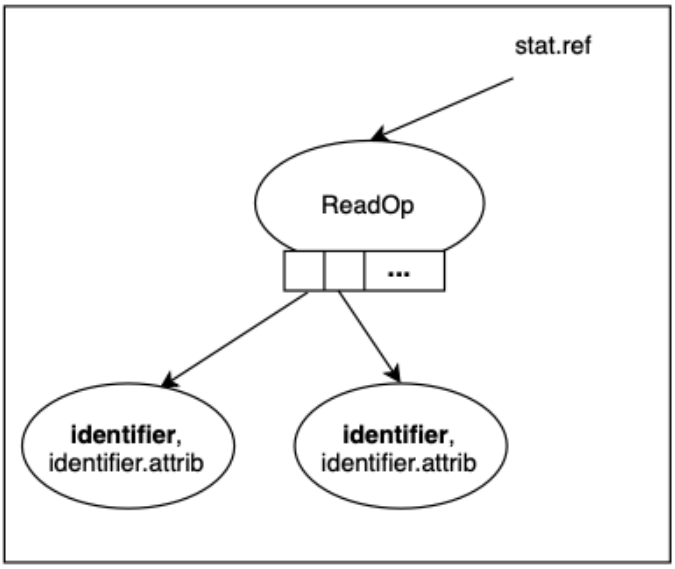




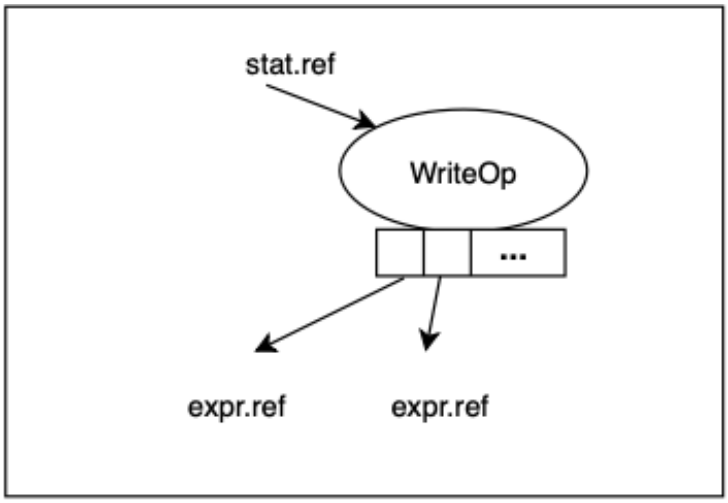
StaList



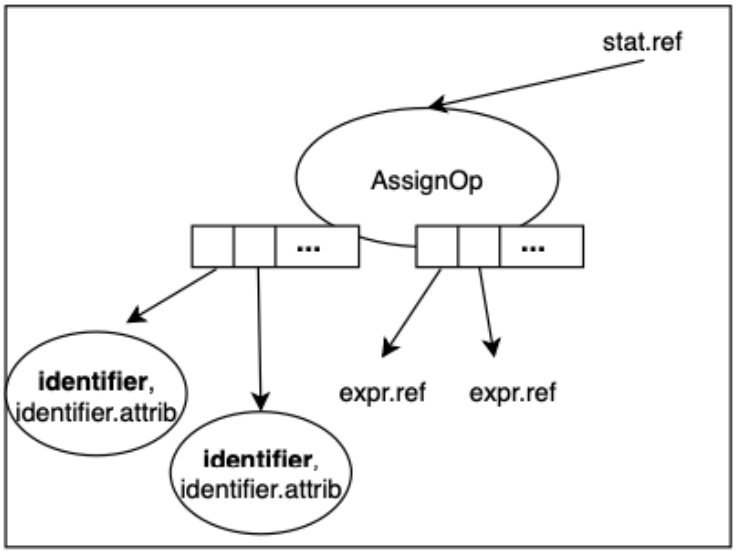
ReadInStat



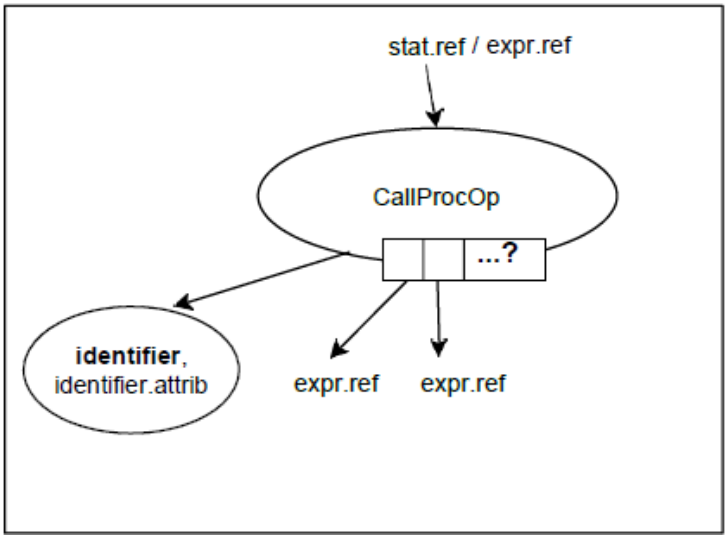
WriteStat



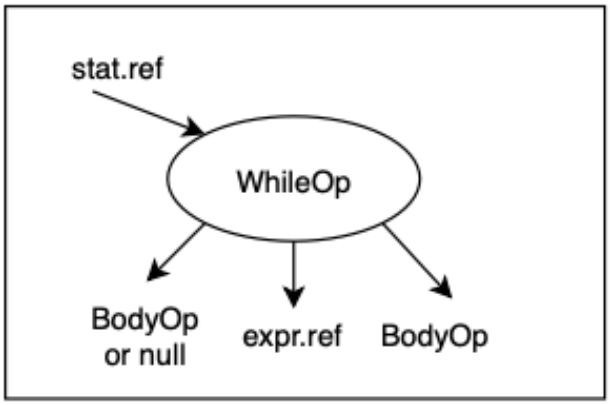
AssignStat



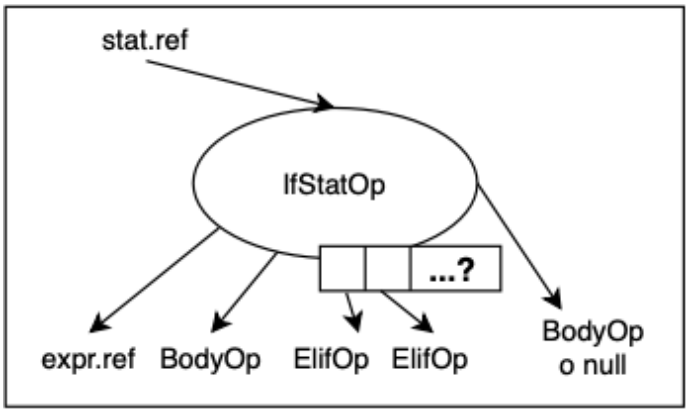
CallProc

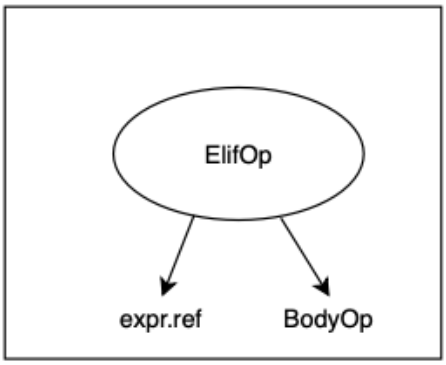


WhileStat

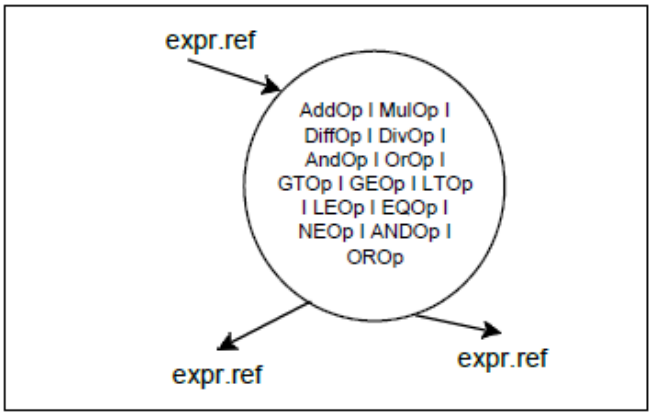


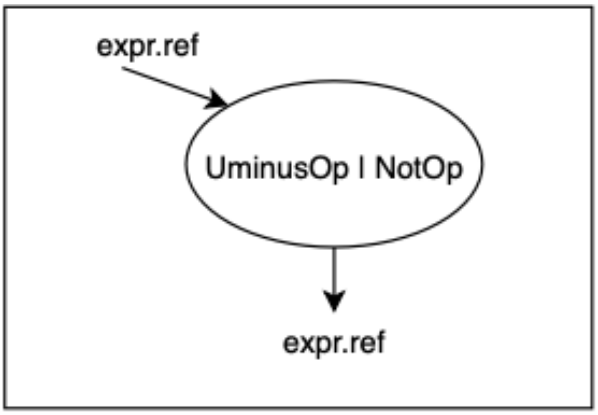
IfStat

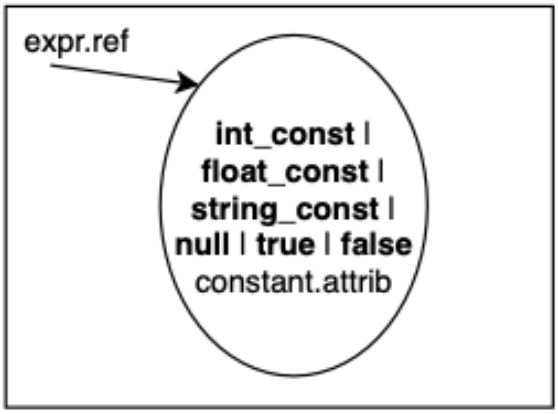

stat.ref
IfStatOp
...?
expr.ref
BodyOp
ElifOp
ElifOp
BodyOp
o null

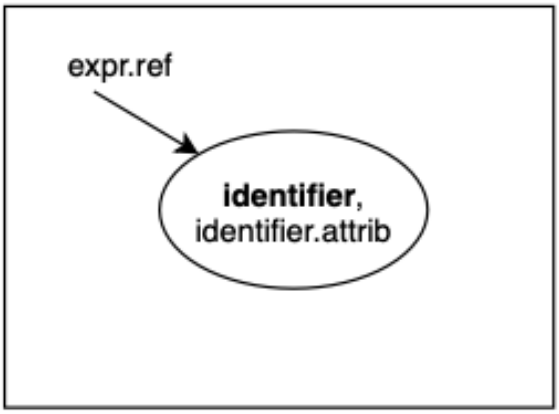

ElifOp
expr.ref
BodyOp

Expr


expr.ref
AddOp | MulOp |
DiffOp | DivOp |
AndOp | OrOp |
GTOp | GEOp | LTOp
| LEOp | EQOp |
NEOp | ANDOp |
OROp
expr.ref
expr.ref


expr.ref
UminusOp | NotOp
expr.ref


expr.ref
int_const |
float_const |
string_const |
null | true | false
constant.attrib


expr.ref
identifier,
identifier.attrib

3 Scelte implementative

3.1 Pattern per identificare i numeri negativi

pattern "number = (\+|\-)?(([1-9][0-9]*)|(0))" modificato in "number = (([1-9][0-9]*)|(0))" in quanto i numeri negativi verrebbero identificati nell'analisi sintattica e non grazie al terminale Uminus. Si veda, per esempio, l'istruzione "n := n - -1". Il -1 non deve essere restituito come un INT_CONST ma deve essere risolta prima l'operazione di UminusOp andando ad esaurire il secondo meno e poi prendere 1 come int const.

ESATTO

```
<DiffOp>
  <Identifier>n</Identifier>
  <UminusOp>
    <IntConst>1</IntConst>
  </UminusOp>
</DiffOp>
```

ERRATO

```
<DiffOp>
  <Identifier>n</Identifier>
  <IntConst>-1</IntConst>
</DiffOp>
```

3.2 Pattern Java per la visita dei nodi

Abbiamo deciso di gestire la visita dell'AST servendoci di due interfacce: Visitor e Visitable.
Le classi XMLVisitor, SemanticVisitor e CLangVisitor implementano un'interfaccia Visitor che obbliga queste ad implementare un metodo visit(). Queste classi sono quelle che "visitano" l'AST.
Le classi corrispondenti ai nodi dell'AST invece implementano un'interfaccia Visitable che obbliga loro ad implementare un metodo accept(). Queste classi sono quelle "visitate".
Prendendo in esame un nodo che si vuole visitare, si chiama su di esso il metodo accept() passandogli il visitor. All'interno del metodo accept() si va a chiamare il metodo visit del visitor passato fornendogli in input il nodo stesso.

3.3 Rilevazione Commenti e Stringhe non chiuse

Per rilevare l'errore su di un commento non chiuso abbiamo implementa all'interno del file .flex un nuovo pattern chiamato "commentMultilineNotClosed" a cui appunto mancano i caratteri di fine

commento asterisco e backslash (*\). Il pattern, nel file flex, è stato inserito dopo quello del commento corretto per rispettare le precedenze sulla rilevazione dei token.
Per rilevare l'errore di stringa non chiusa ci basiamo sull'assunzione che ogni costante stringa debba essere contenuta tra due apici. Dunque, per tutte le stringhe chiuse correttamente avremo sempre in totale ne file sorgente un numero pari di apici ("). Nel caso in cui tale numero sia dispari, l'ultimo apice verrà rilevato nell'analisi sintattica e ciò significa che nel codice sorgente c'è una stringa costante non chiusa.

3.4 Gestione degli identificatori di variabili e funzione nell'analisi semantica

Per gli identificatori di funzione abbiamo creato la classe IdentifierFunction (che estende Identifier) aggiungendo due array per memorizzare il tipo dei parametri che la funzione accetta in ingresso (ArrayList<ParDeclOp>) e quelli che fornisce in uscita (ArrayList<String>).
L'inserimento degli identificatori delle funzioni avviene con l'aggiunta, nella chiave, delle parentesi '()' per fare in modo che nello scope in cui sono inserite le funzioni (nel nostro caso solo quello globale) possiamo dichiarare anche delle variabili con lo stesso nome e viceversa, così quando inseriamo gli identificatori nella symbolTable, l'id della funzione non andrà a generare un errore di "MultipleDeclarationException" con l'id della variabile.

3.5 Lancio delle Eccezioni

Abbiamo deciso di interrompere l'esecuzione dell'analisi semantica e sintattica quando viene catturata un'eccezione, in quanto si potrebbero generare errori a cascata che dipendono soltanto dal primo errore rilevato.

3.6 Utilizzo funzioni come operando di un'espressione

È possibile effettuare delle operazioni dove una delle due espressioni dell'operazione è una funzione, ma solo se questa funzione restituisce un solo valore.
Gestendo così le operazioni che posso avere tra gli operandi una funzione, quando visitiamo la CallProcOp, settiamo il suo nodeType (utilizzato per il TypeChecking) come il suo tipo di ritorno (se quindi ritorna un solo valore), altrimenti questo viene settato come "FunctionMultiReturn", e di conseguenza non potremo utilizzare tale funzione come operando di una funzione.
Es. a := 5 + somma(5, 5); somma deve restituire un solo valore, altrimenti verrà lanciato un errore.

3.7 Traduzione funzioni da toy a C

Una criticità da superare è il fatto che in toy abbiamo funzioni che possono ritornare più valori. Per tradurre una chiamata a funzione in C ci avvaliamo di variabili temporanee che andranno a sostituire i valori di ritorno. Nella funzione chiamante verranno dunque dichiarate tante variabili temporanee quanti sono i valori di ritorno della funzione chiamata. Queste variabili verranno passate (i loro indirizzi), in aggiunta ai parametri attuali, alla funzione è nel momento del ritorno questa funzione andrà a scrivere i risultati che deve ritornare al chiamante in queste apposite variabili temporanee. Le funzioni in C dunque saranno tutte di tipo "void" data questa gestione per i ritorni.

Gestendo in questo modo i ritorni delle funzioni, è necessario "preparare" le funzioni prima che queste siano chiamate da un'espressione in qualche statement e dare allo statement, che fa uso dei ritorni delle funzioni, le variabili temporanee generate facendoci posticipare la scrittura dello statement nel file "C" di destinazione. (Vedere es-piu-ritorni.toy)

Com'è possibile vedere nell'esempio, a due variabili vengono assegnati i valori di ritorno della funzione "duplica"

```
a, b := duplica(x);
```

Per la traduzione in C bisogna prima "preparare" la funzione duplica:

```
int varTemp0;
int varTemp1;
duplica(x, &varTemp0, &varTemp1);
```

e successivamente assegnare le variabili temporanee in cui si è salvato il valore di ritorno ad 'a' e 'b':

```
a = varTemp0;
b = varTemp1;
```

La traduzione della funzione invece sarà:

Toy	C

proc duplica(int x) int, int :
-> x, x
corp;
	void duplica(int x, int* returnTemp0, int* returnTemp1) {
	*returnTemp0 = x;
	*returnTemp1 = x;
	}

4 Regole di inferenza

4.1 Type Matching

$\Gamma \vdash \textbf{null}: \textbf{null}$	$\Gamma \vdash \textbf{void}: \textbf{void}$	$\Gamma \vdash stringa: \textbf{string}$	$\Gamma \vdash decimale: \textbf{float}$
$\Gamma \vdash intero: \textbf{int}$	$\Gamma \vdash \textbf{true}: \textbf{boolean}$	$\Gamma \vdash \textbf{false}: \textbf{boolean}$	
$\dfrac{\Gamma \vdash e: \tau_1 \quad optype1(op_1, \tau_1) = \tau}{\Gamma \vdash (op_1\ e) : \tau}$	$\dfrac{\Gamma \vdash e_1: \tau_1 \quad \Gamma \vdash e_2: \tau_2 \quad optype2(op_2, \tau_1, \tau_2) = \tau}{\Gamma \vdash (e_1\ op_2\ e_2) : \tau}$		
$\dfrac{\Gamma \vdash f : {\tau_i}^{i \in 1\ldots n} \rightarrow {\tau_j}^{j \in 1\ldots m} \quad \Gamma \vdash e_i : {\tau_i}^{i \in 1\ldots n}}{\Gamma \vdash f({e_i}^{i \in 1\ldots n}) : {\tau_j}^{j \in 1\ldots m}}$	$\dfrac{(x_i{}^{i \in 1\ldots n} : \tau) \in \Gamma}{\Gamma \vdash {x_i}^{i \in 1\ldots n} : \tau}$		

4.2 Regole di tipizzazione dei costrutti (Controllo se sono ben tipati)

$\dfrac{\Gamma \vdash stmt_1 \quad \Gamma \vdash stmt_2}{\Gamma \vdash stmt_1\ ;\ stmt_2\ ;}$	$\dfrac{\Gamma \vdash e : \textbf{boolean} \quad \Gamma \vdash stmt_1 \quad \Gamma \vdash stmt_2}{\Gamma \vdash \textbf{while}\ stmt_1 -> e\ \textbf{do}\ stmt_2\ \textbf{od}}$
$\dfrac{\Gamma \vdash e : \textbf{boolean} \quad \Gamma \vdash {elif_i}^{i \in 1\ldots n} \quad \Gamma \vdash stmt_1 \quad \Gamma \vdash stmt_2}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ stmt_1\ {elif_i}^{i \in 1\ldots n}\ \textbf{else}\ stmt_1\ \textbf{fi}}$	
$\dfrac{\Gamma \vdash e : \textbf{boolean} \quad \Gamma \vdash stmt}{\Gamma \vdash \textbf{elif}\ e\ \textbf{then}\ stmt}$	$\dfrac{(x_i{}^{i \in 1\ldots n} : {\tau_i}^{i \in 1\ldots n}) \in \Gamma}{\Gamma \vdash \textbf{readln}({x_i}^{i \in 1\ldots n})}$
$\dfrac{(x_i{}^{i \in 1\ldots n} : {\tau_i}^{i \in 1\ldots n}) \in \Gamma \quad \Gamma \vdash {e_i}^{i \in 1\ldots n} : {\tau_i}^{i \in 1\ldots n}}{\Gamma \vdash {x_i}^{i \in 1\ldots n} := {e_i}^{i \in 1\ldots n}}$	
$\dfrac{\Gamma \vdash f : {\tau_i}^{i \in 1\ldots n} \rightarrow \textbf{void} \quad \Gamma \vdash e_i : {\tau_i}^{i \in 1\ldots n}}{\Gamma \vdash f({e_i}^{i \in 1\ldots n})}$	$\dfrac{(e_i{}^{i \in 1\ldots n} : {\tau_i}^{i \in 1\ldots n}) \in \Gamma}{\Gamma \vdash \textbf{write}({e_i}^{i \in 1\ldots n})}$

5 Tabelle operazionali

BinaryOp			
op_2	type e_1	type e_2	type result
+ - * /	int	int	int
+ - * /	int	float	float
+ - * /	float	int	int
+ - * /	float	float	float
&& ||	boolean	boolean	boolean
<= < = <> > =>	boolean	boolean	boolean
<= < = <> > =>	int	int	boolean
<= < = <> > =>	int	float	boolean
<= < = <> > =>	float	int	boolean
<= < = <> > =>	float	float	boolean
<= < = <> > =>	string	string	boolean

UnaryOp		
op_1	type e_1	type result
-	int	int
-	float	float
!	boolean	boolean