Analisi semantica di Grammo

Questo documento descrive in modo discorsivo le scelte implementative adottate nella fase di **analisi semantica** del progetto *Grammo*. L'obiettivo della fase è trasformare un AST sintatticamente valido in un programma **semanticamente coerente**, intercettando errori come uso di identificatori non dichiarati, incompatibilità di tipo, firme di funzione non rispettate, regole sui return, vincoli su I/O e condizioni booleane nei costrutti di controllo. Tali aspetti sono volutamente lasciati "liberi" dalla grammatica e demandati alla semantica/type checking.

Architettura complessiva: AST + tabella dei simboli + visitor

La fase semantica si basa su tre componenti principali:

1. **Un AST esplicito**, con nodi tipizzati per dichiarazioni, statement ed espressioni (definiti in *ast_nodes.py*).
2. **Una tabella dei simboli a scope impilati**, per registrare funzioni/variabili e risolvere i riferimenti lessicali (definita in *symbol_table.py*).
3. **Un analizzatore semantico in stile visitor**, che attraversa l'AST, costruisce/verifica l'ambiente dei simboli e calcola/controlla i tipi delle espressioni (implementato in *semantic_analyzer.py*).

Questa separazione consente di mantenere:

- **Costruzione dell'AST** (fase sintattica) distinta da **verifica semantica** (fase successiva).
- Una verifica centralizzata e uniforme delle regole, con un punto unico per definire "compatibilità di tipo", "risoluzione di simboli" e "controllo delle firme".

Rappresentazione dell'AST e implicazioni semantiche (*ast_nodes.py*)

L'AST modella in modo esplicito:

- **Dichiarazioni**: *VarDecl* (dichiarazione tipata di uno o più identificatori), *VarInit* (dichiarazione con inizializzazione a costante) e *FuncDef* (funzioni/procedure).
- **Statement**: assegnamento, chiamata procedurale, blocchi, if/elif/else, while, for, I/O, return.
- **Espressioni**: letterali (con tipo intrinseco), riferimenti a variabile, operatori binari/unari, chiamate funzione.

Una scelta importante è distinguere **chiamate usate come espressioni** (nodo *FuncCallExpr*) da **chiamate usate come statement** (nodo *ProcCallStmt*): sintatticamente la forma è la stessa, ma semanticamente il linguaggio impone vincoli diversi (funzione non-void vs procedura void). Questa distinzione è un requisito esplicito della specifica semantica.

Nota: i nodi includono campi per posizione (line, column), utili per error reporting; l'analizzatore li considera quando disponibili per arricchire i messaggi d'errore.

Costruzione dell'AST con attenzione alla semantica (*ast_builder.py*)

Il builder (*ast_builder.py*) produce l'AST a partire dell'albero di parsing Lark e compie alcune scelte che semplificano le verifiche semantiche successive:

Inferenza "strutturale" delle dichiarazioni var

La grammatica prevede sia:

- dichiarazioni tipate (*var int: a, b;*)
- dichiarazioni con inizializzazione a **costante** (*var x = 10;*)

Il builder costruisce due nodi distinti (*VarDecl* e *VarInit*) e, nel caso *VarInit*, porta nel nodo letterale l'informazione di tipo (es. int, real, string, bool) dedotta dalla costante lessicale. Questo rende diretta l'inferenza di tipo nella semantica.

Modellazione di *#(Expr)* negli I/O

La specifica distingue tra:

- espressioni "libere" (tipicamente prompt stringa in input; stringhe o valori in output)
- segmenti "interpolati" *#(Expr)* (target in input; valore da stampare in output).

Per preservare questa informazione nell'AST, il builder rappresenta *#(Expr)* come una forma di **unario speciale** con operatore *#*, mantenendo l'operando come espressione. Questa codifica consente alla semantica di distinguere in modo affidabile "prompt" vs "interpolazione/target" senza dover ricostruire pattern sintattici complessi.

Tabella dei simboli e regole di visibilità (*symbol_table.py*)

La gestione dei simboli è basata su una pila di dizionari (uno per scope), con:

- *enter_scope / exit_scope* per gestire l'ingresso/uscita da un contesto lessicale.
- *insert* per registrare un simbolo nello scope corrente.
- *lookup* per risolvere un identificatore cercandolo **dallo scope più interno verso l'esterno**.

Scelte di scoping

Coerentemente con la specifica, l'analisi semantica implementa un modello in cui:

- esiste uno scope globale
- all'ingresso di una funzione si apre uno scope locale
- i blocchi annidati non introducono scope separati (coerente con "unico livello di scope per le variabili" e divieto di shadowing).

Divieto di shadowing e unicità degli identificatori

Il vincolo "no shadowing" è applicato in modo sostanziale: prima di inserire un nuovo simbolo (variabile, parametro o funzione), l'analizzatore verifica che non esista già un omonimo risolvibile tramite lookup. Ciò impedisce collisioni non solo nello scope corrente, ma anche tra:

- variabili globali e variabili locali
- funzioni e variabili
- parametri e simboli già presenti.

Analizzatore semantico (semantic_analyzer.py)

L'analizzatore usa un **visitor dinamico**: dato un nodo, la procedura *visit* invoca il metodo specifico *visit_<NomeNodo>*. Questo schema rende esplicite le regole semantiche per ogni costrutto e facilita estensioni future.

Strategia multi-pass a livello di programma

L'analisi del *Program* segue una strategia a passaggi:

Passo 1: registrazione firme e simboli globali

- tutte le funzioni vengono registrate in tabella **solo come firme** (nome, tipi parametri, tipo di ritorno);
- le variabili globali vengono validate e inserite.

Questo consente chiamate a funzioni indipendenti dall'ordine di dichiarazione, come previsto dalla specifica sintattica.

Controllo entry point (*main*): Dopo la registrazione, viene imposto il vincolo:

- esistenza di *main*
- *main* deve essere *void*
- *main* non deve avere parametri.

Passo 2: analisi dei corpi funzione

Per ogni funzione:

- si apre uno scope locale
- si inseriscono i parametri come variabili locali (con divieto di collisione)
- si visita il blocco *corpo* con un contesto che memorizza il tipo di ritorno atteso.

Dichiarazioni di variabili e inferenza di tipo

- ***VarDecl***: inserisce una o più variabili con tipo esplicito, rifiutando ridichiarazioni.
- ***VarInit***: inserisce una variabile con tipo dedotto dalla costante (il builder ha già annotato la costante con un *type_name*). Questo implementa l'inferenza richiesta per *var ID = Const;*.

Assegnamenti

Un assegnamento è valido se:

1. il target è un identificatore dichiarato;
2. il simbolo risolto è una variabile (non una funzione);
3. il tipo dell'espressione a destra è compatibile con il tipo della variabile a sinistra.

La compatibilità adotta una regola semplice e intenzionale:

- identità di tipo sempre ammessa;
- promozione implicita **int → real** ammessa;
- nessun'altra conversione implicita (es. real → int, string ↔ numeri, bool ↔ numeri) è accettata.

Funzioni/procedure, chiamate e argomenti

La semantica applica i vincoli richiesti dalla specifica:

- **Chiamata come statement (*ProcCallStmt*)**: ammessa solo se la firma risolta ha *return_type* = *void*. Chiamare una funzione non-void "ignorandone" il risultato è considerato errore nel linguaggio.
- **Chiamata come espressione (*FuncCallExpr*)**: ammessa solo se la firma risolta ha *return_type != void*, perché deve produrre un valore tipizzato utilizzabile in espressione.

In entrambi i casi viene controllato:

- numero di argomenti = numero parametri
- tipo di ciascun argomento compatibile con il tipo del parametro (usando la stessa nozione di compatibilità, inclusa int → real).

Regole sui return e correttezza del flusso di ritorno

Per ogni return:

- se la funzione è void, è ammesso return; ma è vietato *return expr*;
- se la funzione è non-void, è obbligatorio *return expr*; e il tipo dell'espressione deve essere compatibile col tipo di ritorno dichiarato.

Oltre al controllo "locale" sul singolo statement, viene eseguita un'analisi di flusso (*_check_all_paths_return*) per imporre che **tutti i possibili percorsi di esecuzione** restituiscano un valore nelle funzioni non-void. La regola è implementata in modo conservativo:

- un if garantisce return solo se ha else e se *then*, *else* e tutti gli *elif* garantiscono return;
- while e for non sono considerati garanzia di ritorno (la condizione potrebbe essere falsa all'ingresso).

Questa scelta è coerente con l'obiettivo di evitare falsi negativi sulla mancanza di return, e con il vincolo descritto nella specifica.

Controlli di flusso: condizioni booleane

Le condizioni dei costrutti:

- if / elif
- while
- condizione del for

devono risultare di tipo bool. Se la condizione del for è omessa, viene trattata semanticamente come true (quindi valida).

Type checking delle espressioni

Il type checking implementa le regole principali richieste dalla specifica, respingendo combinazioni che la grammatica accetta per scelta progettuale (es. 3 + true, "ciao" < 5, ecc.).

In sintesi:

- **Aritmetica (+ - * /)**: ammessa su numeri; mix int/real produce real (promozione). Inoltre + è ammesso anche tra string e string con risultato string.
- **Logica (&& ||)**: ammessa solo su bool con risultato bool.
- **Confronti**:
  - == e <> richiedono tipi confrontabili (in pratica: stesso tipo o compatibilità numerica con promozione);
  - < <= > >= richiedono operandi numerici (int/real), con risultato bool.
- **Unari**:
  - ! su bool
  - - su int o real.

Vincoli semantici sugli I/O

La semantica degli I/O segue le regole esplicitate nella specifica:

- **Output (<<, <<!)**
  - gli argomenti vengono semplicemente visitati per garantire che ogni espressione sia ben tipata;
  - l'interpolazione *#(Expr)* viene trattata come espressione da validare, distinguendola dai segmenti non interpolati. In questa fase non viene imposto che tutto sia già string: l'idea è demandare la conversione a stringa alla fase di esecuzione/generazione, come previsto dalle "conversioni automatiche" richieste a livello di progetto.
- **Input (>>)**
  - ogni argomento *#(Expr)* deve essere un **identificatore assegnabile**: semanticamente viene richiesto che l'operando sia un riferimento a variabile e che tale variabile esista.
  - gli argomenti "liberi" (non preceduti da #) sono trattati come prompt e devono avere tipo string.

Questa separazione realizza esattamente la distinzione prevista dalla sintassi estesa degli I/O e dai relativi vincoli semantici.

Gestione degli errori semantici

Gli errori semantici sono gestiti con un'eccezione dedicata e un metodo centralizzato (error) che:

- costruisce un messaggio coerente e uniforme;
- quando disponibili, allega informazioni di posizione (linea) del nodo.

La scelta implementativa è "fail-fast": alla prima violazione rilevata, l'analisi termina. Questo semplifica il controllo del flusso e riduce ambiguità diagnostiche; l'infrastruttura è comunque

predisposta per un’eventuale evoluzione verso la raccolta di più errori in una singola esecuzione.