Università degli Studi di Salerno

Dipartimento di Informatica

Corso di Ingegneria dei Linguaggi di Programmazione

Grammo

Implementazione del compilatore per il linguaggio di programmazione Grammo

Salvatore Di Martino

NF22500114

Grammo — Relazione riassuntiva del progetto

Obiettivo e contesto

Il progetto d'esame consiste nella progettazione e realizzazione di un **compilatore completo** per un nuovo linguaggio di programmazione, chiamato **Grammo**. Il compilatore è implementato in **Python** e segue una pipeline classica: **analisi lessicale e sintattica**, **costruzione di un AST intermedio**, **analisi semantica e type checking**, **generazione di codice target LLVM IR**, **ottimizzazione**, ed **esecuzione JIT** tramite **MCJIT**.

L'obiettivo complessivo è trasformare un programma sorgente Grammo (*.gm*) in un **modulo LLVM IR** eseguibile, mantenendo una separazione netta tra fasi (parsing/AST, semantica, codegen) e demandando a ciascuna fase responsabilità precise.

Architettura generale e organizzazione del progetto

Il progetto è organizzato in modo modulare, coerente con il flusso di un compilatore:

- ***lex_syntax/***: grammatica Lark e parsing (codice sorgente → albero di parsing).
- ***ast_builder.py***: trasformazione dell'albero di parsing in **AST strutturato**.
- ***semantic/***: analisi semantica e type checking (AST → AST validato).
- ***codegen/***: generazione LLVM IR, ottimizzazione e JIT execution.
- ***main.py***: entry point CLI che orchestra l'intera pipeline.
- ***test/***: programmi di esempio (*.gm*) per validare i costrutti del linguaggio.
- ***requirements.txt***: file contenente le dipendenze esterne principali (*lark, llvmlite*).

Questa struttura riflette una scelta progettuale precisa: rendere ogni fase indipendente, testabile e sostituibile, riducendo l'accoppiamento tra parsing, regole semantiche e backend LLVM.

Analisi lessicale e sintattica (Lark)

Grammo è definito tramite una grammatica Lark che descrive:

- struttura del programma come sequenza di dichiarazioni top-level (funzioni/procedure e variabili globali);
- sistema di tipi primitivi (***int, real, bool, string***) e tipo speciale **void** per procedure;
- costrutti principali: dichiarazioni, assegnamenti, chiamate, blocchi, if/elif/else, while, for, return, I/O con operatori dedicati.

Una scelta rilevante è che la grammatica **accetta intenzionalmente** molte combinazioni che potrebbero essere "illogiche" dal punto di vista dei tipi (es. operazioni tra tipi incompatibili): la responsabilità di rifiutarle viene spostata alla fase semantica. In questo modo la sintassi rimane espressiva, mentre la correttezza è garantita dal type checking sull'AST.

Inoltre, la sintassi non impone vincoli come l'esistenza di main, la compatibilità dei tipi negli assegnamenti o l'uso corretto delle chiamate (statement vs expression): tali vincoli sono esplicitamente demandati all'analisi semantica.

Costruzione dell'AST intermedio

Dopo il parsing, il compilatore costruisce un **Abstract Syntax Tree esplicito** composto da nodi tipizzati (dichiarazioni, statement, espressioni). L'AST è il fulcro dell'intero progetto: rappresenta una forma intermedia stabile su cui operano sia la semantica sia la generazione di codice.

L'AST builder incorpora anche alcune scelte utili a "preparare" la fase semantica:

- distingue dichiarazioni **tipate** (*VarDecl*) da dichiarazioni con **inferenza da costante** (*VarInit*), propagando nel nodo *literal* l'informazione di tipo dedotta;
- rappresenta il costrutto *#(Expr)* degli I/O come un unario speciale (operatore #), preservando l'informazione necessaria per verifiche semantiche affidabili su prompt vs target/interpolazione.

Analisi semantica e type checking

La fase semantica trasforma un AST sintatticamente valido in un programma **semanticamente coerente**, intercettando errori di:

- uso di identificatori non dichiarati;
- collisioni tra nomi (divieto di shadowing);
- incompatibilità di tipo;
- firme di funzione non rispettate;
- regole sui return;
- vincoli sui costrutti di I/O e condizioni booleane nei controlli di flusso.

Componenti chiave

La semantica si basa su tre pilastri:

1. **AST tipizzato** (nodi e metadati di posizione, utili per error reporting);
2. **tabella dei simboli** a scope impilati (lookup dal più interno verso l'esterno);
3. **visitor semantico** (*visit_<Nodo>*) che attraversa l'AST, gestisce simboli e calcola/valida i tipi.

Strategia multi-pass e vincoli globali

L'analisi del programma adotta una strategia a passaggi:

- prima registra **firme delle funzioni** e **simboli globali**, consentendo chiamate indipendenti dall'ordine di dichiarazione;
- poi analizza i **corpi funzione** in un contesto che conosce il tipo di ritorno atteso.

Viene inoltre imposto il vincolo di entry point:

- deve esistere main;
- main deve essere void;
- main non deve avere parametri.

Regole di compatibilità dei tipi

La compatibilità è deliberatamente semplice e controllabile:

- tipo identico sempre valido;
- promozione implicita ***int → real*** ammessa;
- nessun'altra conversione implicita accettata.

Queste regole vengono applicate coerentemente su assegnamenti, return e parametri/argomenti nelle chiamate.

Chiamate: statement vs espressione

Un vincolo semantico esplicito è la distinzione tra:

- ***ProcCallStmt***: chiamata usata come statement, ammessa solo per void;
- ***FuncCallExpr***: chiamata usata come espressione, ammessa solo per funzioni non-void.

Return e controllo dei percorsi

Oltre al controllo locale del singolo return, è implementata una verifica conservativa "tutti i percorsi ritornano" per funzioni non-void (ad esempio: un if garantisce return solo se ha else e tutti i rami garantiscono return).

Vincoli sugli I/O

La semantica differenzia:

- **output**: le espressioni devono essere ben tipate; la conversione a stringa può essere demandata a fasi successive;
- **input**: i segmenti *#(Expr)* devono essere **assegnabili** (variabili), mentre gli argomenti non preceduti da # fungono da prompt e devono essere *string*.

Generazione del codice LLVM IR (llvmlite)

La code generation assume un AST già **validato semanticamente** e produce un **modulo LLVM IR** coerente con tali vincoli. L'architettura è nuovamente in stile visitor e mantiene stato di generazione (module, builder, funzione corrente, symbol table locale).

Mappatura dei tipi e modello di memoria

I tipi Grammo vengono mappati in LLVM come segue:

- *int → i32, real → double, bool → i1, string → i8*, void → void*.

Il modello per le variabili locali è basato su *alloca* + *load/store*, demandando a LLVM le ottimizzazioni SSA (promozione a registri, eliminazione ridondanze).

Runtime minimale (*libc*)

Per I/O e gestione stringhe, il backend dichiara e usa funzioni esterne (es. *printf, scanf, strlen, strcpy, strcat, malloc*), risolte poi dal JIT.

Espressioni, casting e stringhe

- Operazioni numeriche generano istruzioni intere o floating-point, con bilanciamento a double quando necessario.
- La promozione ***int → real*** è implementata in modo puntuale (assegnamenti, return, argomenti di chiamata) per riflettere esattamente la compatibilità definita in semantica.
- La concatenazione di stringhe con + è implementata tramite *malloc* e *libc* (*strlen/strcpy/strcat*), producendo una nuova C-string heap-allocated.

Control flow

I costrutti if/elif/else, while, for vengono tradotti in basic block LLVM con salti condizionali/incondizionati e blocchi di merge, rispettando la struttura dei flussi.

Ottimizzazione LLVM

Dopo la generazione dell'IR, il progetto applica una pipeline di ottimizzazione basata sul **New Pass Manager**: parsing/verify del modulo, creazione del target machine e applicazione di una pipeline guidata da parametri (es. livello di ottimizzazione).

Questa fase è coerente con la filosofia del backend: produrre IR corretto e semplice (anche "naïve"), lasciando a LLVM il compito di migliorare l'efficienza.

Esecuzione JIT (MCJIT)

L'ultima fase del compilatore esegue il programma **interamente in memoria**, senza generare un eseguibile standalone. In particolare, il backend LLVM:

- inizializza il **target nativo** della piattaforma di esecuzione;
- risolve i **simboli esterni** necessari (funzioni della *libc* come *printf, scanf, malloc*, ecc.), registrandoli esplicitamente quando richiesto;
- compila il modulo LLVM IR tramite **MCJIT**, ne finalizza la traduzione in codice macchina e invoca la funzione di ingresso main, assumendo la firma *void main()*.

Questa scelta architetturale chiude il ciclo *"compila ed esegui"* all'interno dello stesso processo e consente un workflow rapido e interattivo per il test dei programmi Grammo, senza passaggi intermedi di linking o generazione di file binari.

Interfaccia CLI e workflow d'uso

L'interazione con il compilatore avviene tramite **riga di comando**, utilizzando come entry point il file *main.py*, che coordina sequenzialmente tutte le fasi della pipeline.

Il flusso tipico di esecuzione è il seguente:

- parsing del file sorgente e (opzionalmente) stampa dell'AST;
- analisi semantica;
- generazione del codice LLVM IR;
- applicazione delle ottimizzazioni;
- esecuzione JIT del programma compilato.

Sintassi generale

python -m src.grammo.main FILE_SORGENTE [OPZIONI]

Comandi e flag disponibili

- **FILE_SORGENTE (argomento posizionale)**
  Percorso al file sorgente Grammo (*.gm*) da compilare ed eseguire. Questo argomento è obbligatorio e rappresenta l'input principale del compilatore.
- **-o, --output**
  Specifica il percorso di un file in cui salvare il **codice LLVM IR generato**. Il flag è opzionale ed è pensato principalmente per scopi di debug, ispezione del codice intermedio o verifica delle ottimizzazioni applicate. In assenza di questo flag, il codice LLVM viene mantenuto solo in memoria.
- **-O, --opt-level**
  Imposta il **livello di ottimizzazione LLVM**, con valori ammessi da 0 a 3. Un valore più alto abilita una pipeline di ottimizzazioni più aggressive. Il valore di default è 3, coerente con un'esecuzione orientata alle prestazioni.
- **-a, --ast**
  Richiede la **stampa a video della struttura dell'Abstract Syntax Tree (AST)** generato dopo il parsing. La stampa avviene solo se l'analisi sintattica ha successo ed è utile per il debugging del front-end e per la verifica della correttezza della grammatica e dell'AST builder.

Esempio operativo (dalla root del progetto):

python -m src.grammo.main src/grammo/test/input/factorial.gm -o out.ll -a

In questo scenario, l'utente ottiene sia la rappresentazione dell'AST sia il file IR per debug, e il programma viene eseguito richiedendo input da terminale.

Considerazioni finali e limitazioni note

L'implementazione privilegia chiarezza e coerenza tra specifica semantica e backend LLVM. In particolare:

- la semantica è "fail-fast" (termina al primo errore) con messaggi uniformi e, quando disponibili, informazioni di posizione;
- alcune scelte del backend sono funzionali e pragmatiche ma introducono limitazioni:
  - assenza di short-circuiting per && e || (valutazione diretta su *i1*);
  - gestione della memoria delle stringhe basata su *malloc* senza *free* automatico (potenziali leak in programmi long-running o con concatenazioni frequenti);
  - input string con buffer fisso (256 byte) e nuova allocazione per ogni lettura.

Nel complesso, il progetto realizza un compilatore end-to-end: dalla definizione formale della sintassi, alla modellazione in AST, alla verifica semantica rigorosa, fino alla produzione ed esecuzione di codice LLVM IR ottimizzato tramite infrastruttura standard LLVM.