

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

Generazione ed esecuzione del codice target (LLVM)

Questa sezione descrive l'ultima fase della pipeline del compilatore **Grammo**: a partire dall'AST già validato semanticamente (tipi, scoping, firme, vincoli I/O), viene generato codice **LLVM IR**, optionalmente ottimizzato, e infine eseguito tramite **JIT**. La fase di generazione assume quindi che i controlli descritti nel documento di analisi semantica siano già stati effettuati e che l'AST sia coerente.

Ruolo della code generation nel compilatore

Dopo parsing e costruzione dell'AST, l'analisi semantica verifica vincoli di scoping, tipizzazione e correttezza dei costrutti (incluso il vincolo su main: esistenza, void, zero parametri). Il code generator assume quindi un AST “semanticamente valido” e produce un **modulo LLVM IR** coerente con tali regole.

La fase finale è articolata in tre componenti:

1. **CodeGenerator**: AST → LLVM IR (*llvmlite.ir*).
2. **GrammoOptimizer**: ottimizzazione IR con LLVM Pass Manager (*llvmlite.binding*).
3. **JITExecutor**: compilazione ed esecuzione immediata tramite **MCJIT**, con risoluzione simboli esterni (*libc*).

Architettura: Visitor, Module/Builder e simboli

Visitor sull'AST

CodeGenerator implementa una visita per tipo di nodo (metodi *visit_<Nodo>*), mantenendo lo stato corrente di generazione:

- *module*: il contenitore LLVM globale del programma.
- *builder*: *IRBuilder* posizionato nel basic block corrente (presente solo durante la generazione del corpo di una funzione).
- *current_func*: funzione LLVM in generazione.
- *func_symtab*: tabella simboli per variabili locali/parametri (mappa nome → puntatore *alloca*).

Contesto globale vs contesto funzione

La distinzione “globale”/“locale” è realizzata in modo operativo: se **builder** è **None**, il generatore si considera nel contesto globale; altrimenti sta generando dentro una funzione. Questa scelta guida la produzione di:

- **global variables**: *ir.GlobalVariable* nel modulo.
- **local variables**: *alloca* nello stack frame della funzione, tracciate in *func_symtab*.

Mappatura dei tipi Grammo → LLVM

La mappatura è definita in una tabella (*type_map*) ed è coerente con le specifiche semantiche del linguaggio:

- *int* → *i32*

- $real \rightarrow double$
- $bool \rightarrow i1$
- $string \rightarrow i8^*$ (C-string)
- $void \rightarrow void$

Questa scelta implica che:

- le stringhe sono gestite come **puntatori a buffer di caratteri null-terminati** (compatibili con *libc*);
- i booleani vengono trattati come interi a 1 bit, con eventuale estensione quando richiesto da chiamate variadiche (ossia varargs, es. printf).

Dichiarazione e uso della *libc* (runtime minimale)

Il generatore dichiara nel modulo LLVM le funzioni esterne usate come “runtime”:

- $printf(i8^*, ...)$ e $scanf(i8^*, ...)$ per I/O.
- $strlen(i8^*)$, $strcpy(i8^*, i8^*)$, $strcat(i8^*, i8^*)$ per gestione stringhe.
- $malloc(i64)$ per allocazioni heap (buffer input e concatenazioni).

La risoluzione concreta dei simboli avviene lato JIT (sezione 8): su alcune piattaforme (in particolare Windows) i simboli *libc* vengono **registrati esplicitamente** tramite `llvm.add_symbol`.

Variabili e memoria: modello basato su *alloca* + *load/store*

Variabili globali

Le dichiarazioni globali (*VarDecl* e *VarInit* in contesto globale) generano *ir.GlobalVariable* inizializzate a:

- 0 per interi/bool,
- per real,
- *null* per string (puntatore nullo).

Variabili locali e parametri

All’interno delle funzioni:

- ogni variabile locale è allocata con *alloca* e inizializzata (store del valore di default o dell’iniziatore);
- ogni parametro formale viene reso “mutabile” allocando uno slot con *alloca* e copiandovi il valore passato.

Razionale (SSA demandato all’ottimizzatore)

Il generatore produce un IR “naïve” ricco di load/store; la conversione a forma più efficiente (promozione a registri SSA, eliminazione di accessi ridondanti) è demandata alla pipeline di ottimizzazione LLVM.

Generazione di funzioni e blocchi

Definizione funzione e blocco di ingresso

Per ogni *FuncDef*:

1. si costruisce il *FunctionType* (ret + parametri),
2. si crea la *Function* nel modulo,
3. si appende un basic block entry,
4. si inizializza *IRBuilder(entry)*,
5. si resetta *func_symtab* e si alloca lo storage per i parametri.

Return implicito e correttezza dei flussi

Al termine della visita del corpo funzione:

- se la funzione è void e l'ultimo blocco non è terminato, viene emesso ret void;
- se la funzione è non-void e manca una terminazione, viene emesso unreachable.

Questa scelta è coerente con l'aspettativa che l'analisi semantica garantisca già la presenza di return in tutti i percorsi per funzioni non-void; unreachable agisce come fallback/guardia IR.

Espressioni: aritmetica, confronti, logica, chiamate e stringhe

Letterali e riferimenti a variabili

- I letterali numerici/booleani diventano *ir.Constant*.
- Le stringhe letterali diventano **global constant** (array di *i8* con terminatore \0) e vengono restituite come *i8** via *bitcast*.
- Un *VarRef* genera load dal puntatore ottenuto dalla symbol table locale o dalle globali del modulo.

Implementazione rilevante: i letterali stringa sono **deduplicati** (mappa contenuto → global), evitando duplicazioni nel modulo.

Operazioni binarie numeriche e confronti

Per *BinaryExpr*:

- se almeno un operando è *double*, entrambi sono bilanciati a double (con *sitofp* per l'eventuale *i32*) e si usano *fadd/fsub/fmul/fdiv, fcmp_ordered*.
- altrimenti si usano *add/sub/mul/sdiv* e *icmp_signed* per confronti.

Casting隐式 (coercion) int → real

È implementato in tre punti principali:

- **assegnamento** a variabile *real*: inserisce *sitofp* se il valore è *i32*;
- **return** di funzione *real*: inserisce *sitofp* se il valore è *i32*;
- **chiamata funzione**: per ciascun argomento, se il parametro atteso è *double* e il valore è *i32*, inserisce *sitofp*.

Questa politica rispecchia la compatibilità definita dall'analisi semantica (promozione implicita consentita solo in questa direzione).

Operazioni su stringhe: concatenazione con *malloc* + *libc*

Se entrambi gli operandi sono puntatori (interpretati come stringhe C) e l'operatore è +, il code generator produce:

1. *len1 = strlen(lhs), len2 = strlen(rhs)*
2. *total = len1 + len2 + 1*
3. *new_str = malloc(total)*
4. *strcpy(new_str, lhs)*
5. *strcat(new_str, rhs)*
6. restituisce *new_str* come risultato dell'espressione.

Nota tecnica: questa scelta rende la concatenazione **heap-allocated** e, nello stato attuale, non è previsto alcun free automatico (quindi è possibile accumulare memoria in programmi che concatenano intensivamente).

Operatori booleani && e ||

Gli operatori && e || sono generati con *and*_e *or*_su *i1*. Questo equivale a una valutazione "bitwise" su booleani e **non implementa short-circuiting** tramite control flow dedicato.

Statement: assegnamenti, I/O, chiamate, return

Assegnamento

AssignStmt:

- genera il valore RHS,
- risolve il puntatore della variabile (locale o globale),
- applica *sitofp* se necessario (caso *real <- int*),
- store nel puntatore.

Return

ReturnStmt:

- se presente un'espressione, la valuta e gestisce l'eventuale *sitofp* coerente col tipo di ritorno della funzione;
- emette *ret <val>* oppure *ret void*.

Chiamate come statement vs espressione

La distinzione prevista dall'AST (procedure call statement vs function call expression) si riflette direttamente in IR:

- *ProcCallStmt*: call e risultato ignorato.
- *FuncCallExpr*: call e risultato usato come valore in espressione.

Output: *printf* con format string

OutputStmt genera, per ogni argomento:

- se $i32$: $\text{printf}("%d", \text{val})$
- se $double$: $\text{printf}("%f", \text{val})$
- se $i1$: estensione a $i32$ (zext) e $\text{printf}("%d", \text{val_zext})$
- se $i8^*$: $\text{printf}("%s", \text{ptr})$

Se l'output richiede newline ($<<!$), viene stampata anche " \ln ".

Input: *scanf* con gestione speciale delle stringhe

InputStmt tratta gli argomenti in base alla loro natura:

- se l'argomento è un letterale (usato come prompt), viene stampato via printf (come output).
- se è un *VarRef*, si legge in base al tipo della variabile:
 - *int*: $\text{scanf}("%d", \&\text{var})$
 - *real*: $\text{scanf}("%lf", \&\text{var})$
 - *string*: viene allocato un buffer heap da 256 byte (*malloc(256)*), poi $\text{scanf}("%255s", \text{buf})$ e infine si memorizza *buf* nella variabile (store del puntatore).

Coerenza con la semantica di $\#(\text{Expr})$:

- il codice “unwrap” di *UnaryExpr('#', operand)* in input elimina il marker e lavora sull'operando (variabile/letterale), in linea con l'uso semantico di $\#(\dots)$ come indicatore di variabile assegnabile nell'I/O.

Control flow: if/elif/else, while, for

If / Elif / Else con basic blocks

L'*IfStmt* genera un insieme di basic block:

- *if_then* per il ramo *then*,
- un blocco “*next_branch*” e, per ciascun *elif*, una coppia di blocchi (*elif_i_then*, *elif_i_next*) quando necessario,
- *if_merge* come blocco di ricongiungimento.

I salti sono realizzati con *cbranch* per le condizioni e branch incondizionati verso il merge per i rami che non terminano (non *ret/unreachable*).

While

WhileStmt genera:

- *while_cond* → valuta condizione e *cbranch* a *while_body* o *while_end*,
- *while_body* → visita corpo e ritorno a *while_cond* se non terminato,
- *while_end* → continuazione del flusso.

For

ForStmt genera:

- (opzionale) generazione dell'inizializzazione,
- *for_cond* → se la condizione esiste, *cbranch* a *body* o *end*; altrimenti loop infinito (branch diretto al *body*),
- *for_body* → corpo,
- (opzionale) *update*,
- salto a *for_cond* se il blocco non è terminato,
- *for_end* → continuazione.

Ottimizzazione LLVM (New Pass Manager)

GrammoOptimizer.optimize applica una pipeline LLVM basata sul **New Pass Manager**:

1. converte *llvmlite.ir.Module* in *ModuleRef* tramite *parse_assembly(str(module))*;
2. esegue *verify()* per validare l'IR;
3. crea *TargetMachine* dall'host (*from_default_triple*);
4. configura *PipelineTuningOptions* con *speed_level* (default 3) e *size_level* (default 0);
5. crea *PassBuilder* e ottiene un *ModulePassManager*;
6. esegue *mpm.run(mod_ref, pass_builder)* e restituisce il modulo ottimizzato.

Effetto pratico atteso: l'IR generato con *alloca* e accessi load/store può essere riscritto in forma più efficiente (ad es. promozione a registri) dalla pipeline standard LLVM, migliorando prestazioni senza complicare la codegen.

Esecuzione JIT (MCJIT) e linking dei simboli esterni

JITExecutor.run(module_ref) esegue il programma compilato interamente in memoria:

1. inizializza target nativo e *asmprinter* LLVM;
2. crea *TargetMachine* dall'host;
3. tenta di caricare la *libc*:
 - a. su Windows, prova *ctypes.cdll.msvcrt*;
 - b. su altri sistemi (macOS/Linux), cerca esplicitamente la libreria C utilizzando *ctypes.util.find_library('c')*;
 - c. altrimenti *ctypes.CDLL(None)* come fallback;
4. se disponibile, registra esplicitamente gli indirizzi di *printf*, *scanf*, *malloc*, *strlen*, *strcpy*, *strcat* con *llvm.add_symbol* (passaggio critico per la risoluzione dei simboli in JIT su alcune piattaforme);
5. crea l'engine *create_mcjit_compiler(module_ref, target_machine)* e finalizza (*finalize_object*);
6. recupera l'indirizzo della funzione *main* e la invoca come *void main()* tramite *ctypes.CFUNCTYPE(None)*.

Questa strategia è coerente con il vincolo semantico su *main* e consente un ciclo di sviluppo “compila ed esegui” immediato.

Considerazioni finali e limitazioni

Limitazioni note (derivate dall'implementazione attuale)

- **Nessuno short-circuiting** per `&&` e `||`: la semantica di valutazione è quella di and/or su *i1* senza controllo di flusso dedicato.
- **Gestione memoria stringhe**: concatenazioni e input string allocano heap tramite `malloc` senza `free`; in assenza di Garbage Collection o gestione esplicita, sono possibili memory leak nei programmi long-running o con concatenazioni ripetute.
- **Input string**: buffer fisso 256 byte (con `%255s`), scelta prudente contro overflow ma limitante per stringhe più lunghe; inoltre ogni `scanf` su stringa alloca un nuovo buffer.
- **Stampa booleani**: l'implementazione effettiva stampa 0/1 con `%d` (non “true/false”).