Documentazione Tecnica: Minilang Compiler con Backend LLVM

Angelo Barone
Matricola: NF22500157
Corso: Ingegneria dei Linguaggi di Programmazione
Professore: Gennaro Costagliola

GitHub

Anno Accademico 2025/2026

Indice

1 Introduzione

Il progetto ha come scopo la definizione di un linguaggio di programmazione minimale ma robusto e la realizzazione del relativo compilatore in linguaggio Python. La pipeline di compilazione è stata sviluppata interamente "da zero": le fasi di analisi lessicale e sintattica non si avvalgono di generatori automatici, ma implementano rispettivamente un Lexer manuale e un Parser ricorsivo-discendente per garantire il massimo controllo sul processo.

Il backend del compilatore si interfaccia con l'infrastruttura **LLVM** tramite la libreria `llvmlite`, generando codice intermedio (IR) che permette un'alta efficienza e portabilità su diverse architetture.

Oltre alla traduzione del codice sorgente, il sistema include fasi di analisi e trasformazione avanzate tipiche dei compilatori moderni:

- **Ottimizzazione AST:** Implementazione di tecniche di *Constant Folding* (calcolo preventivo delle espressioni costanti), *Dead Code Elimination* (rimozione del codice irraggiungibile) e Semplificazione Algebrica.
- **Analisi Semantica:** Validazione rigorosa del codice tramite costruzione della Symbol Table, gestione dello Scope (di tipo *Flat*) e verifica della coerenza nelle chiamate a funzione (*Arity Check*).
- **Desugaring e Lambda Lifting:** Una fase di trasformazione che normalizza l'AST convertendo costrutti di alto livello (cicli `repeat`, operatore pipe `|>` e funzioni anonime *Lambda*) in strutture standard gestibili dal backend
- **Foreign Function Interface (FFI):** Supporto nativo per la dichiarazione e l'invocazione di funzioni esterne definite in C (come `printf`), permettendo l'interazione con il sistema operativo per operazioni di I/O.

2 Specifiche del Linguaggio

2.1 Specifiche Sintattiche

La struttura sintattica è definita formalmente dalla seguente grammatica libera dal contesto (BNF). L'implementazione è affidata a un parser di tipo **Ricorsivo Discendente**.

La gerarchia delle produzioni per le espressioni (`Expr`) è strutturata per gestire implicitamente la priorità degli operatori. Il parser risolve le espressioni partendo da quelle a priorità inferiore (come `PipeExpr` e `AssignExpr`) scendendo ricorsivamente verso quelle a priorità maggiore (come `MulExpr` e `Primary`).

In presenza di ambiguità, come la distinzione tra un assegnamento e un'espressione logica o tra parentesi raggruppanti e definizioni di Lambda, il parser utilizza tecniche di *Lookahead* per determinare la regola corretta da applicare.

```
// --- Struttura Generale ---
Program     ::= Decls
Decls       ::= Decl Decls
              | /* empty */
Decl        ::= FuncDecl
              | VarDecl
              | ExternDecl
```

```
8
9  // --- Dichiarazioni ---
10 VarDecl     ::= LET ID ASSIGN Expr SEMI
11 ExternDecl  ::= EXTERN FUNC ID LPAR Params RPAR SEMI
12 FunctionDecl    ::= FUNC ID LPAR Params RPAR LBRACE Stmts RBRACE
13 Params      ::= ID COMMA Params
14               | ID
15               | /* empty */
16
17 // --- Statement e Controllo Flusso ---
18 Stmts       ::= Stmt Stmts
19               | /* empty */
20 Stmt        ::= VarDecl
21               | Expr SEMI
22               | RETURN Expr SEMI
23               | IfStmt
24               | WhileStmt
25               | RepeatStmt
26
27 IfStmt      ::= IF LPAR Expr RPAR LBRACE Stmts RBRACE ElseBlock
28 ElseBlock   ::= ELSE LBRACE Stmts RBRACE
29               | /* empty */
30 WhileStmt   ::= WHILE LPAR Expr RPAR LBRACE Stmts RBRACE
31 RepeatStmt  ::= REPEAT LPAR Expr RPAR LBRACE Stmts RBRACE
32
33 // --- Espressioni (Gerarchia di Precedenza) ---
34 Expr        ::= PipeExpr
35
36 // Pipe e Assegnamento (Priorita' Minima)
37 PipeExpr    ::= AssignExpr PIPE PipeExpr
38               | AssignExpr
39 AssignExpr  ::= ID ASSIGN LogicExpr
40               | LogicExpr
41
42 // Operatori Logici e Confronto
43 LogicExpr   ::= EqualityExpr AND LogicExpr
44               | EqualityExpr OR LogicExpr
45               | EqualityExpr
46 EqualityExpr::= RelExpr EQ RelExpr
47               | RelExpr NE RelExpr
48               | RelExpr
49 RelExpr     ::= AddExpr LT AddExpr
50               | AddExpr GT AddExpr
51               | AddExpr LE AddExpr
52               | AddExpr GE AddExpr
53               | AddExpr
54
55 // Aritmetica
56 AddExpr     ::= MulExpr PLUS AddExpr
57               | MulExpr MINUS AddExpr
58               | MulExpr
59 MulExpr     ::= UnaryExpr TIMES MulExpr
60               | UnaryExpr DIV MulExpr
61               | UnaryExpr
62
63 // Operatori Unari e Primari (Priorita' Massima)
64 UnaryExpr   ::= MINUS UnaryExpr
65               | NOT UnaryExpr
```

```
                | Primary
Primary     ::= INTEGER_CONST
                | ID
                | Lambda
                | LPAR Expr RPAR
                | Call

// --- Chiamate e Lambda ---
Call        ::= ID LPAR Args RPAR
Args        ::= Expr COMMA Args
                | Expr
                | /* empty */
Lambda      ::= LPAR Params RPAR ARROW Expr
```

Listing 1: Grammatica BNF del linguaggio

2.2 Specifiche Lessicali

Il Lexer converte il flusso di caratteri in token significativi. Gli spazi bianchi (spazi, tabulazioni e ritorni a capo) vengono ignorati e fungono da separatori.

Di seguito sono definiti i token riconosciuti, suddivisi per categoria. Per gli identificatori e i numeri interi vengono utilizzate espressioni regolari (Regex) per descriverne il pattern.

```
// --- Keywords (Parole Riservate) ---
LET             "let"
FUNC            "func"
EXTERN          "extern"
RETURN          "return"
IF              "if"
ELSE            "else"
WHILE           "while"
REPEAT          "repeat"

// --- Identificatori e Letterali ---
// Deve iniziare con una lettera, seguita da lettere, numeri o
   underscore
ID              [a-zA-Z][a-zA-Z0-9_]* INTEGER_CONST   [0-9]+

// --- Operatori Aritmetici ---
PLUS            "+"
MINUS           "-"
TIMES           "*"
DIV             "/"

// --- Operatori di Confronto ---
LT              "<"
GT              ">"
LE              "<="
GE              ">="
EQ              "=="
NE              "!="

// --- Operatori Logici ---
AND             "&&"
OR              "||"
NOT             "!"
```

```
33
34 // --- Operatori Speciali e Assegnamento ---
35 ASSIGN          "="
36 PIPE            "|>"
37 ARROW           "=>"
38
39 // --- Delimitatori e Punteggiatura ---
40 LPAR            "("
41 RPAR            ")"
42 LBRACE          "{"
43 RBRACE          "}"
44 COMMA           ","
45 SEMI            ";"
```

Listing 2: Definizione dei Token

2.3 Definizione Semantica

L'analisi semantica è responsabile della validazione logica del programma. Il compilatore adotta una strategia a **due passaggi** (*Two-Pass Strategy*): una prima scansione effettua il censimento delle funzioni registrandone le firme, mentre la seconda analizza i corpi per validare le istruzioni.

Le regole semantiche implementate sono le seguenti:

- **Tipizzazione:** Il linguaggio utilizza una tipizzazione dinamica implicita; il tipo delle variabili è determinato al primo assegnamento. A livello di backend LLVM, per semplificare la generazione del codice, tutti i valori numerici e booleani sono rappresentati uniformemente come interi a 64 bit (`i64`).
- **Scope Piatto (Flat Scope):** Ogni funzione definisce un unico ambiente di visibilità (Scope). A differenza di linguaggi come C o Java, le variabili dichiarate all'interno di blocchi annidati (come `if` o `while`) non sono locali al blocco, ma rimangono visibili per il resto della funzione.
- **Arity Check:** Durante la compilazione viene verificata la corrispondenza tra il numero di argomenti passati in una chiamata (`CallExpr`) e il numero di parametri definiti nella firma della funzione. Questo controllo si applica sia alle funzioni utente che a quelle esterne (FFI).
- **Controllo di Definizione:** L'analizzatore impedisce l'uso (lettura o assegnamento) di variabili non ancora dichiarate. Ogni utilizzo di un identificatore viene confrontato con il `current_scope` popolato dalle istruzioni `let` precedenti.

3 Architettura e Implementazione

Il compilatore è strutturato secondo una classica pipeline sequenziale. Ogni fase trasforma la rappresentazione del codice sorgente, passando da una stringa grezza fino all'emissione del codice intermedio. L'architettura software fa ampio uso del **Visitor Pattern** (classe `NodeVisitor` in `ast_nodes.py`) per separare la struttura dei dati (AST) dagli algoritmi che vi operano (Ottimizzazione, Semantica, Generazione Codice).

3.1 Analisi Lessicale (Lexer)

Il modulo `lexer.py`, in combinazione con le definizioni in `tokens.py`, implementa lo scanner che converte il flusso di caratteri in una sequenza di token. L'analisi avviene tramite un automa a stati finiti manuale che gestisce un cursore di posizione (`self.pos`) e il carattere corrente (`self.current_char`).

Le caratteristiche implementative principali sono:

- **Tokenizzazione Manuale:** A differenza di approcci basati su librerie di espressioni regolari precompilate, il Lexer implementa metodi dedicati (`integer()` e `_id()`) che consumano i caratteri sequenzialmente.
  - **Identificatori e Keyword:** Ogni stringa alfanumerica viene inizialmente letta come identificatore; successivamente, viene verificata la sua presenza nel dizionario `KEYWORDS`. Se presente (es. `"let"`, `"func"`, `"repeat"`), viene emesso il token riservato corrispondente, altrimenti viene classificata come `TokenType.ID`.
- **Disambiguazione tramite Lookahead:** Per distinguere operatori che condividono lo stesso prefisso, il Lexer utilizza il metodo `peek()` per ispezionare il carattere successivo senza consumarlo. I casi critici gestiti includono:
  - **Pipe vs OR:** Il carattere `|` attiva un controllo sul successivo: se segue `>`, genera `TokenType.PIPE` (`|>`); se segue `|`, genera `TokenType.OR` (`||`).
  - **Assegnamento vs Uguaglianza vs Lambda:** Il carattere `=` può generare un assegnamento (`=`), un confronto (`==`) o, se seguito da `>`, l'operatore freccia per le lambda (`=>`).
  - **Minore/Maggiore:** Gestione corretta di `<` vs `<=` e `>` vs `>=`.
- **Gestione Robustezza:** Il metodo `skip_whitespace()` elimina spazi, tabulazioni e ritorni a capo tra i token. In caso di caratteri illegali, il metodo `error()` solleva un'eccezione bloccante fornendo il contesto dell'errore per il debugging.

3.2 Analisi Sintattica (Parser)

Il modulo `parser.py` trasforma il flusso lineare di token prodotto dal Lexer in una struttura gerarchica ad albero (*Abstract Syntax Tree*). L'implementazione segue rigorosamente la grammatica BNF definita, adottando un approccio **Ricorsivo Discendente**.

Le principali caratteristiche architetturali includono:

- **Gestione del Flusso dei Token:** Il parser si avvale di metodi utilitari per navigare il flusso:
  - `peek(offset)`: Ispeziona i token successivi senza consumarli (Lookahead).
  - `consume(type)`: Verifica e consuma il token atteso, avanzando il cursore o sollevando un errore di sintassi.
  - `match(types...)`: Controlla se il token corrente appartiene a un set di tipi accettati e, in caso affermativo, lo consuma.

- **Gerarchia Implicita delle Precedenze:** La priorità degli operatori non è definita in una tabella esterna, ma è codificata nell'ordine di chiamata dei metodi. I metodi per le espressioni a bassa priorità chiamano quelli ad alta priorità, facendo sì che questi ultimi vengano valutati prima (essendo più vicini alle foglie dell'albero).
  - Ordine di chiamata: `Pipe` → `Assign` → `Logic` → `Equality` → `Relational` → `Additive` → `Multiplicative` → `Unary` → `Primary`.
- **Risoluzione delle Ambiguità (Lookahead):** Alcuni costrutti sintattici richiedono di guardare oltre il token corrente per determinare la regola corretta da applicare:
  - **Lambda vs Raggruppamento:** Poiché sia le espressioni tra parentesi `(x)` che le definizioni di lambda `(x) => ...` iniziano con `LPAREN`, il parser utilizza il metodo `is_lambda_lookahead()` per cercare la presenza del token freccia (`=>`) prima di decidere quale nodo costruire.
  - **Assegnamento vs Espressione:** Un identificatore a inizio riga può indicare un assegnamento o l'uso di una variabile. Il metodo `parse_assign_expr` controlla il token successivo per distinguere `x = ...` (Assegnamento) da `x + ...` (Espressione Logica).
- **Tipologia dei Nodi AST:** I nodi sono definiti in `ast_nodes.py` utilizzando le `@dataclass` per garantire immutabilità. Esiste una distinzione netta tra `Stmt` (nodi che eseguono azioni, come `IfStmt` o `WhileStmt`) ed `Expr` (nodi che producono valori, come `BinaryExpr`), facilitando le fasi successive di analisi semantica e generazione codice.

3.3 Ottimizzazione (Optimizer)

Il modulo `optimizer.py` implementa un trasformatore di AST basato sul **Visitor Pattern**. L'ottimizzatore attraversa l'albero con una strategia **Bottom-Up** (dalle foglie alla radice), garantendo che le sotto-espressioni siano semplificate prima di valutare i nodi genitori. Questo permette di ottimizzare espressioni complesse annidate (es. $(3+2)*0 \rightarrow 5*0 \rightarrow 0$) in un singolo passaggio.

Le tecniche implementate sono:

- **Constant Folding Esteso:** Valuta a tempo di compilazione qualsiasi espressione binaria o unaria i cui operandi siano letterali (numeri).
  - *Aritmetica:* Gestisce le operazioni standard ($+, -, *, /$) e include un controllo di sicurezza per la divisione per zero.
  - *Logica e Confronti:* Valuta operatori relazionali ($<, >, ==, \dots$) e logici ($\&\&, ||$), restituendo 1 (Vero) o 0 (Falso).
  - *Unari:* Pre-calcola negazioni aritmetiche ($-$) e logiche ($!$).
- **Semplificazione Algebrica:** Applica identità matematiche per ridurre il costo computazionale, eliminando operazioni inutili:
  - *Identità della Somma/Sottrazione:* $x + 0 \rightarrow x$, $0 + x \rightarrow x$, $x - 0 \rightarrow x$.
  - *Identità della Moltiplicazione/Divisione:* $x \times 1 \rightarrow x$, $1 \times x \rightarrow x$, $x/1 \rightarrow x$.

- *Elemento Annullatore:* $x \times 0 \to 0$, $0 \times x \to 0$. Sostituisce l'intero sotto-albero con il letterale 0.

- **Dead Code Elimination (DCE):** Rimuove intere porzioni di codice che non verranno mai eseguite perché protette da condizioni staticamente false:
  - *If Statements:* Se la condizione è staticamente vera (! = 0), il nodo `IfStmt` viene sostituito direttamente dal blocco `then_branch`. Se è falsa (0), viene sostituito dal blocco `else_branch` (o rimosso se l'`else` manca).
  - *While Loops:* Se la condizione del ciclo è staticamente falsa (0), l'intero nodo `WhileStmt` viene rimosso dall'AST (restituendo `None`), eliminando il ciclo dal programma finale.

3.4 Analisi Semantica

Il modulo `semantic_analysis.py` implementa la validazione logica del codice tramite la classe `SemanticAnalyzer`. L'analisi adotta una strategia a **doppio passaggio** (*Two-Pass Strategy*) per risolvere le dipendenze in avanti (forward references).

Il processo si articola nelle seguenti fasi:

1. **Censimento Globale (Passo 1):** Il visitatore scansiona inizialmente tutte le dichiarazioni di funzione (`FunctionDecl` e `ExternDecl`) senza analizzarne il corpo.
   - Popola la tabella dei simboli globale `functions_arity` mappando il nome della funzione al numero di parametri attesi. Questo censimento preventivo è fondamentale per supportare l'ordine arbitrario delle definizioni.

2. **Analisi dei Corpi (Passo 2):** Viene effettuata una seconda visita approfondita per validare le istruzioni all'interno delle funzioni.
   - **Gestione dello Scope (Flat Scope):** Viene utilizzato un insieme `current_scope` per tracciare le variabili visibili. I blocchi interni (`if`, `while`) non resettano l'ambiente, rendendo le variabili dichiarate nei blocchi visibili anche all'esterno per il resto della funzione.
   - **Variable Shadowing:** Il linguaggio supporta la ridefinizione di variabili (*Shadowing*). Una nuova dichiarazione `let` con un identificatore già esistente è permessa: essa non sovrascrive il valore della variabile originale, ma ne oscura la visibilità per le istruzioni successive, allocando una nuova istanza.
   - **Controllo di Definizione (Use-before-Declaration):** Viene applicato un controllo rigoroso sull'esistenza delle variabili prima del loro utilizzo:
     - *In Lettura (`VariableExpr`):* Verifica che l'identificatore sia presente nel `current_scope`.
     - *In Scrittura (`AssignExpr`):* Impedisce l'assegnamento a variabili non precedentemente dichiarate con `let`.
   - **Validazione Chiamate a Funzione:** Per ogni nodo `CallExpr`, il compilatore verifica l'esistenza della funzione e applica l'**Arity Check**, confrontando il numero di argomenti forniti con la firma registrata.
   - **Univocità dei Parametri:** Durante la registrazione dello scope locale di una funzione, viene verificato che non esistano nomi di parametri duplicati per evitare ambiguità.

3.5 Desugaring e Lambda Lifting

Il modulo `desugaring.py` normalizza l'AST rimuovendo le strutture complesse ("Sintactic Sugar") per semplificare il backend e renderlo compatibile con LLVM IR. Oltre ai cicli e alle pipe, il modulo gestisce le funzioni anonime tramite il **Lambda Lifting**:

- **Repeat Loop:** Il costrutto `repeat(N)` viene riscritto come un ciclo `while` con gestione automatica di un contatore temporaneo univoco.
- **Pipe Operator (|>):** L'espressione $a \mid> f$ viene trasformata nella chiamata canonica $f(a)$. Se f è una catena di chiamate, l'argomento viene iniettato come primo parametro.
- **Lambda Lifting:** Poiché il backend LLVM supporta solo funzioni dichiarate a livello globale, le espressioni Lambda (es. `(x) => x * 2`) subiscono una trasformazione radicale:
  1. **Estrazione:** L'espressione lambda viene rimossa dal punto in cui è definita.
  2. **Generazione Funzione:** Viene creata una nuova `FunctionDecl` globale con un nome univoco generato automaticamente (es. `__lambda_0`).
  3. **Adattamento del Corpo:** Poiché le lambda sono espressioni, il loro corpo viene avvolto in un `ReturnStmt` all'interno di un blocco.
  4. **Sostituzione:** Nel punto originale, la lambda viene sostituita da una `VariableExpr` che fa riferimento al nome della nuova funzione generata.

  Questa procedura permette al generatore di codice di trattare le lambda come normali chiamate a funzione o puntatori a funzione.

3.6 Generazione Codice (Backend LLVM)

La classe `LLVMCodeGen` (definita in `codegen.py`) orchestra la traduzione finale dall'AST ottimizzato al codice intermedio *LLVM IR* avvalendosi della libreria `llvmlite`.

Le scelte architetturali principali includono:

- **Strategia a Due Passaggi (Two-Pass Generation):** Per risolvere le dipendenze in avanti (come chiamate a funzioni definite successivamente o ricorsione), la generazione è divisa in due fasi: prima vengono emesse le dichiarazioni (firme) di tutte le funzioni, e successivamente ne vengono generati i corpi .
- **Gestione della Mutabilità e SSA:** LLVM IR utilizza la forma *Static Single Assignment* (SSA), dove i registri virtuali (es. `%1`) sono immutabili e possono essere assegnati una sola volta. Per supportare le variabili mutabili del linguaggio sorgente senza dover calcolare manualmente complessi grafi SSA (nodi Phi), il compilatore adotta il modello di allocazione sullo stack:
  - **Allocazione:** Ogni variabile dichiarata viene associata a uno spazio sullo stack tramite l'istruzione `alloca`, che restituisce un puntatore. Sebbene il registro contenente l'indirizzo del puntatore sia immutabile, il contenuto della memoria a cui esso punta è modificabile.

  - **Accesso:** Le operazioni di scrittura (assegnamenti) vengono tradotte in istruzioni `store` verso quell'indirizzo, mentre le letture vengono tradotte in istruzioni `load`.
  - Questo approccio delega alle fasi di ottimizzazione successive di LLVM (come `mem2reg`) il compito di promuovere le variabili di stack in registri efficienti.

- **Variable Shadowing:** Il modello basato sullo stack facilita l'implementazione del *Shadowing*. Una ridichiarazione di variabile (nuovo nodo `VarDecl` con nome esistente) genera una nuova istruzione `alloca`, ottenendo un indirizzo di memoria distinto. La symbol table locale (`func_symtab`) viene aggiornata con il nuovo puntatore, "oscurando" il precedente per tutte le istruzioni generate successivamente nello stesso scope.

- **Gestione dei Tipi e Casting:** Il backend uniforma i tipi numerici a 64 bit (`i64`). I risultati booleani dei confronti (che in LLVM sono `i1`) vengono estesi a `i64` tramite `zext` per compatibilità, mentre gli operatori logici sono mappati su istruzioni bitwise (`and_`, `or_`).

- **Traduzione del Flusso di Controllo:** I costrutti di controllo di alto livello sono tradotti in grafi di *Basic Blocks* collegati da salti. Per l'`if-else` e il `while`, vengono generati blocchi specifici per la condizione, il corpo e i punti di unione (Merge Blocks), gestendo il flusso tramite istruzioni `branch` (incondizionato) e `cbranch` (condizionato).

- **Compatibilità ABI:** Viene iniettato l'attributo di funzione `"stack-probe-size"` nel codice IR finale per garantire la corretta esecuzione e compatibilità con l'ABI di sistema su piattaforme specifiche (es. Windows).

3.7 Orchestrazione della Pipeline (Main)

Il modulo `main.py` agisce come *driver* principale del compilatore, coordinando l'esecuzione sequenziale dei vari componenti e gestendo il flusso dei dati dalla lettura del file sorgente fino alla scrittura del codice macchina intermedio.

La funzione `compile_source` implementa la logica di controllo, garantendo che ogni fase riceva l'output della precedente e gestendo eventuali eccezioni specifiche. La pipeline è così strutturata:

1. **Preprocessing e Tokenizzazione:** Il codice sorgente viene letto e passato al `Lexer`, che produce una lista completa di token tramite la funzione di supporto `get_all_tokens`. Questa fase iniziale filtra immediatamente errori lessicali.

2. **Costruzione dell'AST:** La lista di token viene consumata dal `Parser`, che restituisce il nodo radice (`ast_root`) del programma. Se vengono rilevati errori sintattici, il processo si interrompe prima delle fasi successive.

3. **Trasformazione (Desugaring):** Una caratteristica distintiva dell'architettura implementata è l'esecuzione del `Desugarer` immediatamente dopo il parsing. Questa scelta architetturale permette di normalizzare l'AST prima dell'analisi semantica, semplificando le fasi successive che opereranno su un set ridotto di nodi standard.

4. **Validazione Semantica:** Il `SemanticAnalyzer` riceve l'AST trasformato e ne verifica la correttezza logica (scope e arity). Poiché il desugaring è già avvenuto, l'analizzatore lavora su una struttura che riflette più da vicino la logica di esecuzione finale.

5. **Ottimizzazione:** L'`Optimizer` esegue una visita sull'AST validato, applicando *Constant Folding* e pulizia del codice. Poiché le costanti sono state calcolate e i rami morti eliminati, il generatore di codice riceverà un albero più snello.

6. **Generazione e Output:** Infine, il `LLVMCodeGen` traduce l'AST ottimizzato in una stringa rappresentante il codice IR. Il driver si occupa quindi di salvare il risultato su file (default: `output.ll`) e fornisce all'utente le istruzioni per il linking con il runtime C tramite `clang`.

L'utilizzo della libreria `argparse` permette di configurare il compilatore da riga di comando, abilitando opzionalmente flag di `-debug` per ispezionare lo stato interno tra una fase e l'altra.

4 Strategia di Testing

La validazione del compilatore è stata effettuata su due livelli: un'ampia suite di test unitari per verificare i singoli componenti della pipeline e un test di integrazione finale per dimostrare le capacità complete del linguaggio in uno scenario reale (Foreign Function Interface e I/O).

4.1 Unit Testing

È stata sviluppata una suite di test automatici utilizzando il framework `unittest` di Python. Ogni modulo del compilatore dispone di un file di test dedicato per garantire la regressione e la correttezza:

- **Lexer (`test_lexer.py`):** Verifica il corretto riconoscimento di tutti i token, incluse le keyword, gli operatori composti (es. `==`, `=>`, `|>`) e la gestione degli errori lessicali.
- **Parser (`test_parser.py`):** Controlla la costruzione dell'AST, la precedenza degli operatori matematici e la corretta gestione delle strutture sintattiche come dichiarazioni di funzioni e blocchi condizionali.
- **Semantic Analysis (`test_semantic.py`):** Testa il rilevamento di errori semantici critici, quali:
  - Uso di variabili non definite.
  - Violazione dell'Arity Check (numero errato di argomenti nelle chiamate).
  - Duplicazione dei nomi dei parametri.
- **Optimizer (`test_optimizer.py`):** Valuta l'efficacia del *Constant Folding* (calcolo di espressioni costanti), delle identità algebriche e della rimozione del *Dead Code* (blocchi irraggiungibili).

- **Desugaring** (`test_desugaring.py`)**:** Verifica la corretta trasformazione del costrutto `repeat`, dell'operatore pipe e la corretta estrazione delle Lambda in funzioni globali nominate.
- **Code Generation** (`test_codegen.py`)**:** Analizza l'output IR generato, assicurandosi che le istruzioni LLVM (come `alloca`, `store`, `br`) corrispondano alla logica del sorgente.

4.2 Test di Integrazione: Calcolatrice Interattiva

Per soddisfare il requisito di progetto relativo alla creazione di un programma in grado di gestire un menu utente, calcoli e cicli di continuazione, è stato realizzato un test di integrazione composto da due file:

1. `test_menu.mini`: Il codice sorgente nel linguaggio proprietario.
2. `runtime.c`: Un supporto runtime in C per gestire Input/Output.

Questo test dimostra l'uso combinato di tutte le feature del linguaggio: **FFI**, **cicli while**, **costrutti if/else**, **zucchero sintattico (repeat, pipe)** e **chiamate a funzione multipla**.

4.2.1 Codice Sorgente (`test_menu.mini`)

Il programma definisce la logica di business. Importa funzioni C tramite `extern`, implementa una funzione `calculate` per la logica matematica e gestisce il flusso principale nel `main`.

```
extern func show_menu();
extern func get_input();
extern func print_result(n);
extern func print_prompt();
extern func print_dash(n);
extern func print_error(n);

func calculate(op, a, b) {
    if (op == 1) { return a + b; }
    if (op == 2) { return a - b; }
    if (op == 3) { return a * b; }
    if (op == 4) { return a / b; }
    return 0;
}

func main() {
    let running = 1;
    while (running) {
        show_menu();
        let choice = get_input();

        // Uscita dal programma
        if (choice == 5) {
            print_error(-222); // Codice uscita
            running = 0;
        } else {
            // Validazione input
            if (choice > 0 && choice < 5) {
```

```
                print_prompt();
                let val1 = get_input();
                print_prompt();
                let val2 = get_input();

                let res = calculate(choice, val1, val2);

                // Utilizzo operatore Pipe
                res |> print_result;

                // Utilizzo costrutto Repeat (estetico)
                repeat(20) {
                    print_dash(0);
                }
            } else {
                print_error(-9999); // Codice errore
            }
        }
    }
    return 0;
}
```

Listing 3: Logica della calcolatrice in MiniLang

4.2.2 Runtime di Supporto (`runtime.c`)

Poiché il linguaggio compila in LLVM IR ma non dispone di una standard library nativa, le funzioni di I/O sono delegate a C tramite linking dinamico.

```c
#include <stdio.h>
#include <stdint.h>

// Interfaccia per visualizzare il menu
int64_t show_menu() {
    printf("\n--- CALCOLATRICE MINILANG ---\n");
    printf("1. Addizione (+)\n");
    printf("2. Sottrazione (-)\n");
    printf("3. Moltiplicazione (*)\n");
    printf("4. Divisione (/)\n");
    printf("5. Esci\n");
    printf("Scegli un'operazione: ");
    return 0;
}

// Wrapper per scanf
int64_t get_input() {
    long long n;
    scanf("%lld", &n);
    return (int64_t)n;
}

int64_t print_result(int64_t n) {
    printf(">> RISULTATO: %ld\n", n);
    return 0;
}

// ... altre funzioni di utilita' (print_error, print_dash) ...
```

Listing 4: Funzioni di supporto C

4.2.3 Procedura di Esecuzione

Il test viene eseguito seguendo questi passi di compilazione e linking:

```
# 1. Compilazione del runtime C
 gcc -c tests/test_menu/runtime.c -o runtime.o

# 2. Compilazione del sorgente MiniLang in LLVM IR
 python main.py tests/test_menu/test_menu.mini -o output.ll

# 3. Traduzione dell'LLVM IR in codice oggetto
clang --target=x86_64-pc-windows-gnu -c output.ll -o output.o

# 4. Linking dell'eseguibile finale
gcc output.o runtime.o -o programma.exe

# 5. Esecuzione
./programma.exe
```

Questo scenario valida con successo la capacità del compilatore di produrre codice binario, gestendo correttamente lo stack, il passaggio di parametri e il flusso di controllo.

5 Conclusioni e Sviluppi Futuri

Il progetto ha portato alla realizzazione completa di un compilatore per un linguaggio imperativo minimale, partendo dall'analisi lessicale fino alla generazione di codice macchina intermedio. L'architettura modulare, coordinata dal driver principale (`main.py`), ha permesso di isolare le diverse fasi di elaborazione, facilitando il testing e la manutenzione del codice.

5.1 Risultati Raggiunti

Gli obiettivi prefissati in fase di specifica sono stati soddisfatti:

- **Frontend Manuale:** La realizzazione di Lexer e Parser senza l'ausilio di generatori automatici ha permesso un controllo capillare sulla gestione degli errori e sulle regole di precedenza degli operatori.
- **Pipeline di Ottimizzazione:** L'integrazione di un `Optimizer` che agisce sull'AST prima della generazione del codice garantisce che espressioni costanti e rami morti vengano rimossi, riducendo il carico di lavoro a runtime.
- **Robustezza Semantica:** Il sistema di analisi a doppio passaggio ("Two-Pass") assicura la coerenza delle chiamate a funzione e la gestione corretta dello scope, inclusa la validazione dell'Arity Check.
- **Integrazione LLVM e FFI:** Il backend genera codice IR compatibile con l'infrastruttura LLVM, delegando la gestione della memoria allo stack. Il supporto per la *Foreign Function Interface* (FFI) è stato validato con successo tramite l'implementazione di un menu interattivo che invoca funzioni C per l'I/O.

5.2 Analisi Critica delle Scelte Progettuali

Alcune decisioni implementative meritano una riflessione in ottica di bilanciamento tra complessità e funzionalità:

- **Gestione della Memoria (Stack vs Registri):** La scelta di allocare tutte le variabili locali sullo stack tramite `alloca` ha semplificato drasticamente il backend, evitando la necessità di calcolare manualmente la forma SSA. Sebbene questo produca un codice IR più verboso (con molte istruzioni `load`/`store`), si affida efficacemente ai passaggi di ottimizzazione standard di LLVM (come `mem2reg`) per la pulizia finale.
- **Desugaring Preventivo:** La trasformazione di costrutti complessi come `repeat` e `|>` in forme primitive prima dell'analisi semantica ha ridotto la complessità del backend, che deve gestire un set minore di istruzioni.
- **Variable Shadowing:** Il compilatore permette la ridefinizione di variabili nello stesso scope allocando nuova memoria. Se da un lato questo offre flessibilità, dall'altro richiede attenzione da parte del programmatore per evitare la perdita accidentale di riferimenti a variabili precedenti.

5.3 Sviluppi Futuri

Il compilatore, pur essendo funzionale, si presta a diverse estensioni:

1. **Type System Esteso:** Attualmente il backend supporta solo interi a 64 bit (`i64`). Un'evoluzione naturale sarebbe l'introduzione di tipi `float`, stringhe e strutture dati composte.
2. **Analisi del Flusso Dati:** Implementare un'analisi "Reaching Definitions" permetterebbe ottimizzazioni più aggressive e la rilevazione di variabili non inizializzate a tempo di compilazione.
3. **Error Recovery:** Migliorare il parser per supportare il recupero dagli errori, permettendo di rilevare più errori sintattici in una singola compilazione invece di arrestarsi al primo guasto.

In conclusione, il progetto dimostra come la combinazione di tecniche classiche (parsing ricorsivo) e strumenti moderni (LLVM) permetta di costruire un compilatore efficiente ed estensibile.