Programming Language
Csar

SPECIFICHE TECNICHE

Ingegneria dei Linguaggi di Programmazione
2024-2025

Michele Cillo	**NF22500124**
Salvatore De Feo	**NF22500176**

Introduzione

I Csar è un linguaggio di programmazione procedurale imperativo e fortemente tipizzato, ispirato alla sintassi del latino e basato sul celebre linguaggio di programmazione “C”.

Le caratteristiche principali del Csar sono:

- **Sintassi Latina:** tutte le key word sono derivate dal latino (functio, si, dom, principalis)

- **Tipizzazione Forte:** il linguaggio supporta tipi primitivi (integer, littera, boolianus) ovvero (integer, char, boolean) e non ammette conversioni implicite, garantendo la correttezza delle operazioni in fase di analisi semantica

- **Gestione dello Scope:** Csar adotta un modello di scoping a due livelli (Globale e locale) permettendo così la possibilità d’utilizzo della ricorsione e semplificando la gestione della Symbol Table.

- **Interoperabilità:** usando il costrutto “externus”, Csar può invocare funzioni definite in librerie esterne (C Runtime), permettendo operazioni di Input/Output senza dover implementare funzioni di basso livello. (è previsto externus solo per la funzione printf)

- **Case Sensitivity:** il linguaggio distingue tra caratteri maiuscoli e minuscoli. Identificatori come ‘Nome’ e ‘nome’ sono considerati **diversi,** le keywords devono essere scritte in **minuscolo**

Il compilatore è sviluppato in **Python** e adotta un'architettura modulare a pipeline, gestita dal driver principale `main.py`.

1 Pipeline di Compilazione

La compilazione è “orchestrata” dalla funzione compile_file in main.py, che esegue in sequenza le fasi di parsing, trasformazione, analisi semantica e generazione codice.

MAIN.PY

```
def compile_file(filename):
    # 1. Caricamento Grammatica


    # 2. Parsing (Crea Parse Tree)


    # 3. Trasformazione (Crea AST)


    # --- 3b. ANALISI SEMANTICA (Type Checking & Scope) ---


    # 4. CodeGen (Backend)


    # 5. Ottimizzazione (Middle-end)


    # 6. Salvataggio
    print(f"\nSalvataggio: {out_filename} ✅")
```

2 Analisi Lessicale e Sintattica (Frontend)

Il frontend del compilatore Csar è implementato utilizzando la libreria **Lark** in Python. La definizione del linguaggio avviene attraverso un unico file di grammatica (`grammar.lark`) che separa formalmente le regole lessicali (Lexer) da quelle sintattiche (Parser).

2.1 Analisi Lessicale (Lexer)

Il Lexer è responsabile della conversione del flusso di caratteri in ingresso (codice sorgente) in un flusso di **Token**(parole logiche). I token sono definiti tramite Espressioni Regolari (Regex).
Categorie di Token

- **Keywords Riservate:** Ispirate al latino, non possono essere utilizzate come identificatori (es. `functio`, `si`, `dom`, `redde`, `principalis`).

- **Tipi di Dato (Primitivi)**: `integer`, `littera`, `boolianus`, `nullum`.

- **Identificatori (ID):** Definiti dalla regex `/[a-zA-Z][a-zA-Z0-9_]*/`. Devono iniziare con una lettera, seguita da alfanumerici o underscore.
- Letterali:

    - Numeri interi (`NUMBER`): `/[0-9]+/`
    - Caratteri (`CHAR`): Gestiti tra singoli apici, inclusi caratteri di escape (es. `'\n'`).

- **Elementi Ignorati:** Spazi bianchi (`WS`) e commenti su singola riga (`// ...`) vengono scartati dal Lexer e non influenzano l'albero sintattico.

2.2 Analisi Sintattica (Parser)

Il Parser analizza la sequenza di token per costruire il **Parse Tree**, verificando che rispetti la struttura grammaticale del Csar.
Struttura del Programma
La regola di avvio (`start`) definisce la struttura rigida di un file sorgente Csar, che deve seguire quest'ordine preciso per facilitare la compilazione *one-pass*:

1. **Dichiarazioni Esterne (`external_decl_list`):** Prototipi di funzioni C (es. `externus functio print...`).
2. **Dichiarazioni di Funzioni (`function_decl_list`):** Logica del programma definita dall'utente.
3. **Blocco Principale (`main_block`):** L'entry point `principalis`, unico per ogni programma.

2.2.1 Gestione delle Espressioni e Precedenza degli Operatori

La grammatica gestisce la precedenza degli operatori matematici e logici **implicitamente** attraverso una gerarchia di regole annidate. Le regole più "profonde" hanno priorità maggiore (vengono valutate prima).
La gerarchia definita in `grammar.lark` è la seguente (ordine crescente) :

1. **`logic_or` (`||`):** Livello più basso.
2. **`logic_and` (`&&`):** Ha precedenza su OR.
3. **`comp_expr` (Confronti `::`, `<`, `>`):** Valutati prima della logica.
4. **`sum_expr` (Somma/Sottrazione `+`, `-`):** Valutati prima dei confronti.
5. **`term_expr` (Moltiplicazione/Divisione `*`, `/`):** Hanno precedenza sulle somme.
6. **`factor` (Atomici):** Numeri, variabili, chiamate di funzione e parentesi `(expr)`. Le parentesi permettono di scavalcare la precedenza standard risalendo la gerarchia.

```
// ========================================
// GRAMMATICA CSAR (Lark)
// ========================================

start: program

 // ANALISI SINTATTICA PARSER
// STRUTTURA

// --- DICHIARAZIONI ---
// --- PARAMETRI ---
param_list: param ("," param)*
param: type ID

// --- TIPI E FRECCIA ---
// --- BLOCCO --
// --- ISTRUZIONI ---

// --- ESPRESSIONI ---

// --- OPERATORI ---

//LEXER- ANALISI LESSICALE
// --- TOKEN DEFINIZIONI ---
TYPE_INT: "integer"
TYPE_CHAR: "littera"
TYPE_BOOL: "boolianus"
TYPE_NULL: "nullum"
ID: /[a-zA-Z][a-zA-Z0-9_]*/
NUMBER: /[0-9]+/
CHAR: /'([^'\\\n]|\\.)'/
%import common.WS
%ignore WS
COMMENT: /\/\/[^\n]*/
%ignore COMMENT
```

3. Costruzione dell'Albero Sintattico Astratto (AST)

La fase successiva al parsing è la trasformazione del Parse Tree (CST) generato da Lark in un Abstract Syntax Tree (AST). Mentre il Parse Tree rappresenta fedelmente la sintassi (incluse parentesi, virgole e keyword), l'AST è una rappresentazione semplificata e gerarchica focalizzata sulla logica del programma, essenziale per le fasi successive di analisi semantica e generazione del codice.

3.1 Definizione dei Nodi (`ast_nodes.py`)

La struttura dell'AST è definita attraverso una gerarchia di classi Python, implementate come Data Classes (@dataclass) per garantire immutabilità e chiarezza strutturale. Ogni costrutto del linguaggio Csar (funzioni, istruzioni, espressioni) ha un nodo corrispondente e tutti i nodi sono “figli” della classe base comune “Node”.

Gerarchia delle Classi
Tutte le classi ereditano da una classe base comune Node (una classe vuota segnaposto), che permette di trattare polimorficamente qualsiasi nodo dell'albero.
I nodi sono suddivisi in tre categorie logiche:

Nodi Strutturali (Scheletro del Programma)
Definiscono l'organizzazione di alto livello del codice sorgente.

- **ProgramNode:** La radice dell'albero. Contiene tre liste distinte: dichiarazioni esterne, funzioni utente e il blocco main.
- **FunctionNode:** Rappresenta la definizione di una funzione (functio). Conserva nome, parametri, tipo di ritorno e il corpo.
- **ExternNode:** Rappresenta la firma di una funzione esterna (es. C Runtime) importata con externus.
- **BodyNode:** Rappresenta un blocco di codice { ... }.

Nodi di Istruzione (Statements)

Rappresentano azioni che non restituiscono un valore, ma modificano lo stato del programma o il flusso di esecuzione.

- **VarDeclNode:** Dichiarazione di variabile (es. integer x : 5 .)
- **AssignNode:** Assegnamento di un valore a una variabile esistente.
- **IfNode / WhileNode:** Strutture di controllo. Contengono riferimenti ricorsivi ad altri nodi (condition è un'espressione, body è un BodyNode).
- **ReturnNode:** Istruzione redde.
- **ProcCallNode:** Chiamata a una funzione (procedura).

```
@dataclass
class IfNode(Node):
    condition: Node              # L'espressione condizionale
    then_body: 'BodyNode'        # Il blocco da eseguire se
vero
    else_body: Optional['BodyNode'] # Il blocco opzionale
(aliter)
```

Nodi di Espressione (Expressions)
Rappresentano calcoli che producono un valore (aritmetico, logico o booleano).

- **BinaryOpNode:** Rappresenta qualsiasi operazione a due operandi (+, -, *, &&, ::, ecc.).
  - *Attributi:* left (nodo sinistro), op (simbolo operatore), right (nodo destro).
- **LiteralNode:** Rappresenta un valore costante (foglia dell'albero). Contiene il valore grezzo e il suo tipo (es. 5, 'integer').
- **VarExprNode:** L'uso di una variabile all'interno di un calcolo (es. la x in x + 1).
- **FunCallNode:** Chiamata a funzione che restituisce un valore.

```
@dataclass
class BinaryOpNode(Node):
    left: Node
    op: str  # +, -, *, /, ::, <, && ...
    right: Node
```

3.2 Il Trasformatore (transformer.py)

Il passaggio da Parse Tree ad AST è gestito dalla classe CsarTransformer, che eredita da lark.Transformer. Questo componente applica il pattern Visitor in modalità *bottom-up* (dalle foglie verso la radice): per ogni regola grammaticale riconosciuta, viene invocato un metodo specifico che "impacchetta" i token in un nodo AST pulito.

Strategie di Trasformazione Adottate:

Pulizia Sintattica:
Il Transformer scarta automaticamente gli elementi "rumorosi" della grammatica che non servono alla logica, come le parentesi tonde ( ) nelle espressioni, le virgole , nelle liste parametri e le keyword (si, functio, redde).

```
def external_decl(self, items):
    # Struttura: [ID, (Params?), Arrow, ReturnType]
    name = str(items[0])

    # Caso A: Ci sono i parametri (Lunghezza 4)
    if len(items) == 4:
        params = items[1]
        ret_type = items[3]

    # Caso B: NON ci sono parametri (Lunghezza 3)
    else:
        params = []
        ret_type = items[2]

    return ExternNode(name=name, params=params,
return_type=str(ret_type))
```

Gestione delle Liste (items):
I metodi del transformer ricevono un parametro items (una lista di figli processati). Il codice mappa posizionalmente questi elementi sugli attributi dei nodi AST.

```
def block(self, items):
    # items[0]: lista dichiarazioni variabili
    # items[1]: lista istruzioni
    return BodyNode(var_decls=items[0], statements=items[1])
```

Appiattimento delle Espressioni Binarie:
Per gestire la precedenza degli operatori definita nella grammatica (regole a cascata come sum_expr, term_expr), il Transformer utilizza un metodo generico _binary_op. Questo metodo itera sulla lista di token (es. [3, "+", 4, "*", 5]) e costruisce nodi BinaryOpNode annidati, preservando l'associatività a sinistra.

```
def _binary_op(self, items):
    node = items[0]
    # Salta a passi di 2: prende Operatore e Operando Destro
    for i in range(1, len(items), 2):
        op = str(items[i])
        right = items[i+1]
        node = BinaryOpNode(left=node, op=op, right=right)
    return node
```

4. Analisi Semantica (Middle-end)

L'Analisi Semantica è la fase in cui il compilatore verifica la coerenza logica del programma, assicurandosi che rispetti le regole del sistema di tipi e la visibilità delle variabili (scoping), regole che non possono essere catturate dalla sola grammatica Context-Free. Questa fase prende in input l'AST generato dal Transformer e, se non rileva errori, lo restituisce "validato" al generatore di codice. Il componente principale è la classe SemanticVisitor (definita in semantic_visitor.py), che implementa tre controlli fondamentali:

1. **Scope Checking:** Gestione della visibilità dei simboli.
2. **Type Checking:** Verifica della compatibilità dei tipi (Strong Typing).
3. **Flow Checking:** Verifica della coerenza nelle strutture di controllo (es. condizioni booleane).

4.1 Gestione dello Scope (Symbol Table)

Per gestire la visibilità delle variabili, è stata implementata una **Symbol Table** dedicata (class SymbolTable). Il linguaggio Csar adotta un modello di scoping a due livelli:

- **Scope Globale:** Contiene le definizioni di funzioni (functio) e dichiarazioni esterne (externus). È visibile ovunque.
- **Scope Locale:** Contiene i parametri formali delle funzioni e le variabili dichiarate all'interno dei blocchi. È effimero: viene creato all'ingresso di una funzione (enter_function) e distrutto all'uscita (exit_function).

Politica di Shadowing

- **Divieto di Shadowing:** È vietato dichiarare una variabile locale con lo stesso identificatore di una funzione o variabile globale.
- **Divieto di Ridichiarazione:** Non è possibile dichiarare due variabili con lo stesso nome nello stesso scope locale.

```
def define(self, name, type_name, kind):
    if kind == 'local':
        if name in self.global_scope:
            raise SemanticError(f"Errore Shadowing: '{name}'
nasconde una globale.")
        self.local_scope[name] = (type_name, ‘local')
```

4.2 Il Sistema di Tipi (Type System)

Csar è un linguaggio fortemente tipizzato. Il SemanticVisitor percorre l'AST calcolando (inferenza) il tipo di ogni espressione e verificando che coincida con le aspettative.

Le regole implementate sono:

- **Operazioni Aritmetiche (+, -, *, /):** Accettano solo operandi integer e restituiscono integer. Non è permesso sommare caratteri o booleani.
- **Operazioni Logiche (&&, ||):** Accettano solo boolianus.
- **Confronti (::, <, >):** Richiedono che i due operandi siano dello stesso tipo (omogenei), ma restituiscono sempre boolianus.
- **Strutture di Controllo (si, dom):** La condizione tra parentesi *deve* risolversi in un tipo boolianus. Non sono ammesse conversioni implicite (es. if (5) è errore semantico).
- **Coerenza del Return:** Il valore restituito da redde deve coincidere con il tipo dichiarato nella firma della funzione.

4.3 Implementazione Tecnica: Visitor Pattern

Per navigare l'AST, è stato utilizzato il Visitor Pattern. Invece di utilizzare una lunga catena di if/else o switch, l'implementazione sfrutta la riflessione (reflection) di Python per il dispatch dinamico dei metodi.

1. **Modularità:** Ogni nodo ha la sua logica di validazione isolata in un metodo dedicato.
2. **Estensibilità:** Aggiungere un nuovo nodo all'AST richiede solo l'aggiunta del metodo visit_NuovoNodo senza modificare la logica di dispatching.

```
def visit(self, node):
    # Costruzione dinamica del nome del metodo basata sul
tipo del nodo
    method_name = f'visit_{node.__class__.__name__}'
    # Recupero del metodo tramite riflessione
    method = getattr(self, method_name, self.generic_visit)
    return method(node)
```

5. Generazione del Codice Intermedio

La fase di generazione del codice traduce l'AST validato in LLVM IR (Intermediate Representation). Questa rappresentazione è un linguaggio assembly agnostico rispetto all'architettura hardware (RISC-based), tipizzato staticamente e basato sul formato SSA (Static Single Assignment).Il componente responsabile è la classe **CodeGenVisitor** (in codegen.py), che utilizza la libreria llvmlite per costruire il modulo IR.

5.1 Strategia di Allocazione della Memoria

Per semplificare la fase di generazione, il compilatore adotta una strategia "Stack-Only" (o Naive Generation), ogni variabile locale (o parametro) viene allocata nello stack frame della funzione.

Il ciclo di vita di una variabile segue questo pattern:

1. **Dichiarazione:** Si usa l'istruzione alloca per riservare spazio nello stack. Questo restituisce un puntatore.
2. **Assegnamento:** Si usa l'istruzione store per scrivere un valore nell'indirizzo di memoria puntato.
3. **Lettura:** Si usa l'istruzione load per leggere il valore corrente dalla memoria in un registro temporaneo.

5.2 Mapping dei Tipi (Type System Mapping)

I tipi primitivi del Csar vengono mappati direttamente sui tipi nativi di LLVM:

Tipo Csar	Tipo LLVM IR	Descrizione
`integer`	`i32`	Intero a 32 bit con segno.
`littera`	`i8`	Intero a 8 bit (ASCII).
`boolianus`	`i1`	Intero a 1 bit.
`nullum`	`void`	Tipo vuoto per procedure.

5.3 Traduzione del Flusso di Controllo (Control Flow)

if o while vengono tradotti in un grafo di **Basic Blocks** collegati da istruzioni di salto (branch).

- **IF Statement (si):** Viene tradotto generando tre blocchi: then, else (opzionale) e merge.
- **WHILE Statement (dom):** Richiede la creazione di un blocco cond (per valutare la guardia del ciclo), un blocco body e un blocco end.

5.4 Gestione delle Operazioni

Le espressioni dell'AST vengono tradotte in istruzioni aritmetico-logiche:

- Gli operatori matematici usano le varianti con segno (es. sdiv per la divisione).
- Gli operatori di confronto vengono mappati su icmp_signed (Integer Compare Signed).

6. Ottimizzazione (Middle-end)

Il modulo **llvm_optimizer.py** agisce come **Middle-end**, trasformando l'IR grezzo in una versione ottimizzata prima che venga passato al backend finale (Clang/LLVM).

6.1 Architettura del Pass Manager

L'ottimizzazione è gestita tramite le API di llvmlite.binding, configurando un PassManagerBuilder con i seguenti parametri:

- **Optimization Level:** O3 (Massimo livello di ottimizzazione standard).
- **Inlining Threshold:** 100 (Soglia aggressiva per l'inlining delle funzioni).

```
pm_builder = llvm.create_pass_manager_builder()
pm_builder.opt_level = 3  # Livello massimo!
pm_builder.inlining_threshold = 100
```

6.2 Principali Ottimizzazioni Applicate

- **Constant Folding & Propagation**
  L'ottimizzatore valuta le espressioni costanti a tempo di compilazione.
  - *Input:* integer x : 3 + 4 .
  - *Output:* integer x : 7 . (L'istruzione add viene eliminata).
- **Dead Code Elimination (DCE)**
  Rimuove le porzioni di codice che non influenzano l'output del programma o che sono irraggiungibili
- **Function Inlining**
  Le chiamate a funzioni brevi vengono sostituite direttamente con il corpo della funzione chiamata. Questo elimina l'overhead della chiamata (salvataggio registri, salto, ritorno) ed espone ulteriori opportunità di ottimizzazione nel contesto del chiamante.

7. Linking e Generazione Eseguibile

La fase finale della catena di compilazione è il **Linking** (Collegamento).

7.1 Risoluzione dei Simboli Esterni

Quando il compilatore Csar incontra una dichiarazione come:

```
externus functio printf(littera* s) -> nullum .
```

Il Backend LLVM genera una dichiarazione declare nel file IR, ma **non genera il corpo della funzione**. Lascia un "buco" (un riferimento simbolico irrisolto). Il Linker ha il compito di riempire questi buchi cercando le definizioni mancanti nelle librerie di sistema standard.

7.2 Il Processo di Build Completo

la creazione dell'eseguibile richiede l'intervento di un driver di compilazione esterno (**Clang**).
Il flusso logico è il seguente:

1. **Compilazione (Csar)**: `sorgente.csar` → `codice.ll` (LLVM IR).
2. **Assemblaggio (Clang/LLVM):** `codice.ll` → `codice.o` (File Oggetto, codice macchina non eseguibile).
3. **Linking (System Linker):** `codice.o` + `Libreria C Standard` → `programma.exe`.

Comando di Esecuzione:

In questa fase, Clang riconosce che le funzioni externus (come printf) appartengono alla libreria standard del C e le collega automaticamente all'eseguibile finale.

```
clang output.ll -o programma.exe
```