Programming Language
Csar

Ingegneria dei Linguaggi di Programmazione
2024-2025

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

Introduzione

Il 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** utilizzando la libreria **Lark** per l’analisi lessicale e sintattica, e **LLVMLite** per la generazione dell’Intermediate Code.

Specifiche Lessicali

A seguire sono definite le regole per l'analisi lessicale del linguaggio Csar. Tali regole sono utilizzate dall'analizzatore lessicale per leggere il flusso di caratteri (stream) del codice sorgente e raggrupparli in unità significative (Token).

Il Csar utilizza il set di caratteri ASCII standard ed è case-sensitive (distinzione tra maiuscole e minuscole).

I token sono classificati in due categorie principali:

- **Token Statici**: definiti da stringhe fisse (es. principalis, ::)

- **Token Dinamici**: Definiti tramite Espressioni Regolari (Regex), utilizzate per riconoscere pattern complessi (identificatori, numeri, char).

Key Words

Il linguaggio presenta le seguenti parole chiave, che non possono essere utilizzate come identificatori

Categoria	Token (Parola Chiave)	Descrizione
Struttura	principalis	Punto di ingresso del programma (Main).
	functio	Definizione di una funzione.
	externus	Dichiarazione di funzione esterna (es. libreria C).
Tipi di Dato	integer	Numero intero.
	Littera	Carattere singolo (Char)
	boolianus	Valore booleano (Vero/Falso).
	nullum	Tipo nullo (void) o valore nullo.
Controllo Flusso	si	Inizio blocco condizionale (If).
	aliter	Ramo alternativo (Else).
	dom	Ciclo condizionale (While).
	redde	Istruzione di ritorno (Return).
Costanti Logiche	verum	Valore booleano Vero (True).
	falsum	Valore booleano Falso (False).

Operatori e Simboli

I seguenti simboli rappresentano operatori o punteggiatura sintattica

Categoria	Token	Descrizione
Punteggiatura	.	Terminatore di istruzione (Punto)
	:	Operatore di assegnamento o inizializzazione
	->	tipo di ritorno della funzione
	,	Separatore di argomenti/parametri
	( )	Parentesi tonde (raggruppamento espressioni/parametri)
	{ }	Parentesi graffe (delimitatori di blocco)
Aritmetici	+ - * /	Addizione, Sottrazione, Moltiplicazione, Divisione.
Confronto	::	Uguaglianza
	!=	Disuguaglianza
	< <=	Minore, Minore o uguale
	> >=	Maggiore, Maggiore o uguale
Logici	&&	AND logico
	ll	OR logico
	!	NOT logico (negazione)

Token Dinamici

Questi token non sono fissi ma devono seguire un pattern specifico riconosciuto dall’analizzatore lessicale tramite Regex

Nome Token	Espressione Regolare (Regex)	Esempi
ID (Identificatore)	[a-zA-Z][a-zA-Z0-9_]*	somma, x1, Var_2
NUMBER (intero)	[0-9]+	0, 14, 28
CHAR(littera)	/'([^'\\n]|\.)'/	a’, ‘b’
Comment	//[^\n]*	Commenti singola riga (ignorati dal parser)

Esempi Errati	Motivazione
1var	(inizia con numero)
_temp	(inizia con underscore)
3.14	(reali non supportati)
3A	(Non può contenere Lettere)

Specifiche Sintattiche

In questa sezione viene definita formalmente la struttura grammaticale del linguaggio Csar.
L'analisi sintattica (Parsing) prende in input la sequenza di token generata dall'analizzatore lessicale e verifica che rispetti le regole del linguaggio.

LEGENDA DEI SIMBOLI USATI NELLA DESCRIZIONE DELLE PRODUZIONI

```
::=  Definisce una regola di produzione

|    indica un'alternativa (OR)

/* vuoto */   Rappresenta la produzione vuota ed indica che
il non-terminale può non generare nulla
```

L'albero di derivazione risultate da queste regole è la base per la costruzione dell'**Abstract Syntax Tree (AST)** utilizzato nelle fasi successive dell'analisi semantica e generazione del codice.

Struttura del programma

```
Program ::= ExternalDeclList FunctionDeclList MainBlock

ExternalDeclList ::= ExternalDecl ExternalDeclList
                   | /* vuoto */

FunctionDeclList ::= FunctionDecl FunctionDeclList
                   | /* vuoto */
```

Dichiarazione Funzione Esterna

Permette di definire prototipi di funzioni definite in librerie esterne (C runtime)

```
ExternalDecl ::= 'externus' 'functio' ID '(' ParamList ')'
Arrow ReturnType '.'
```

Dichiarazione Funzione

```
FunctionDecl ::= 'functio' ID '(' ParamList ')' Arrow
ReturnType Block

ParamList ::= Param ',' ParamList
            | Param
            | /* vuoto */

Param ::= Type ID

Arrow ::= '->'
ReturnType ::= Type | 'nullum'
Type ::= 'integer' | 'littera' | 'boolianus'
```

Funzione “principale” (Main)

Punto di ingresso del Programma, è necessaria in ogni programma Csar ed è definito dalla Keyword “principalis”

```
MainBlock ::= 'principalis' Block
```

Blocco e Istruzioni

Un blocco è racchiuso tra parentesi graffe

```
Block ::= '{' VarDeclList StmtList '}'

VarDeclList ::= VarDeclStmt VarDeclList | /* vuoto */

StmtList ::= Stmt StmtList | /* vuoto */

Stmt ::= AssignStmt | IfStmt | WhileStmt | ReturnStmt |
ProcCallStmt | Block
```

Dichiarazione Variabili

- Ogni variabile è visibile dalla riga di dichiarazione fino alla fine del blocco dove è definita
- Non è possibile dichiarare due variabili con lo stesso nome all'interno dello stesso scope.

```
VarDeclStmt ::= Type ID OptionalInit '.'

OptionalInit ::= ':' Expr
               | /* vuoto */
```

Assegnamento

Modifica il valore di una variabile esistente (già dichiarata in precedenza).

```
AssignStmt ::= ID ':' Expr '.'
```

Controllo di Flusso (If/While)

```
IfStmt ::= 'si' '(' Expr ')' Block ElseBlock
ElseBlock ::= 'aliter' Block | /* vuoto */

WhileStmt ::= 'dom' '(' Expr ')' Block
```

Return

```
ReturnStmt ::= 'redde' OptionalExpr '.'
OptionalExpr ::= Expr | /* vuoto */
```

Chiamata procedura

Chiamata a funzione quando il valore di ritorno non è specificato

```
ProcCallStmt ::= ID '(' ExprList ')' '.'
```

Espressioni

```
Expr ::= LogicOr

LogicOr ::= LogicAnd ( '||' LogicAnd )*
LogicAnd ::= CompExpr ( '&&' CompExpr )*
CompExpr ::= SumExpr ( CompOp SumExpr )*
SumExpr ::= TermExpr ( AddOp TermExpr )*
TermExpr ::= Factor ( MulOp Factor )*

Factor ::= '(' Expr ')'
         | ID
         | NUMBER
         | LITTERA
         | 'verum' | 'falsum' | 'nullum'
         | FunCall

FunCall ::= ID '(' ExprList ‘)'
ExprList ::= Expr ',' ExprList
           | Expr
           | /* vuoto */



CompOp ::= '::' | '!=' | '<' | '<=' | '>' | '>='
AddOp  ::= '+' | '-'
MulOp  ::= '*' | '/'
```

Funcall: chiamata a funzione quando il valore di ritorno è specificato

SPECIFICHE SEMANTICHE

Le regole che l'analizzatore semantico deve verificare visitando l'AST.

Gestione dello Scope

Il linguaggio Csar adotta una versione dello scope a due livelli che permette di creare funzioni ricorsive:

- **Scope Globale:** contiene i nomi delle funzioni (definite o externus)

- **Scope Locale (Funzione)**: ogni funzione crea un proprio ambiente locale contenente:
  - Variabili dichiarate nel corpo della funzione
  - Parametri formali

- **Scope "Piatto" nei blocchi interni:** all'interno dei controlli di flusso (si, dom) non vengono creati nuovi scope. Una variabile dichiarata all'inizio della funzione è visibile ovunque nella stessa funzione.

- **Divieto di Shadowing**: non è consentito dichiarare una variabile locale con lo stesso identificatore di una variabile globale o di una funzione.

- **Visibilità:** tutte le variabili prima di essere utilizzate devono essere dichiarate.

All'interno dei controlli di flusso **non** vengono creati nuovi scope per i blocchi interni (si, dom).
Una variabile dichiarata all'interno di una struttura di controllo è visibile in tutto il resto della funzione.

Regole di Tipo

Il linguaggio Csar è fortemente tipizzato:

- **Dichiarazioni**: **non** è possibile ridichiarare una variabile con lo stesso nome all'interno dello stesso scope

- **Inizializzazione**: se presente (Type ID : Expr .), il tipo di Expr deve coincidere esattamente con Type

- **Type Checking delle Espressioni**

    - **Operatori Aritmetici** (+ , - , * , /) richiedono operandi di tipo **integer** e restituiscono **integer**

    - **Operatori Logici** (&& , ||) richiedono operandi di tipo boolianus e restituiscono boolianus

    - **Operatori di Confronto** (:: , < , <= , > , >=) restituiscono boolianus

- **Valore di ritorno**: L'espressione "redde" deve coincidere con il tipo dichiarato nella firma della funzione

Input/Output

L'interazione con l'utente avviene tramite funzioni esterne definite nel runtime C e importate con externus.
Il compilatore verifica solo la correttezza dei tipi dei parametri passati alle funzioni esterne.

Esempio di Codice in Csar

Funzione Fattoriale ricorsiva in Csar

```
// Dichiarazione funzioni esterne
externus functio scribe_int(integer x) -> nullum .
externus functio scribe_char(littera c) -> nullum .

functio fattoriale(integer n) -> integer {
    si (n :: 0) {
        redde 1 .
    } aliter {
        redde n * fattoriale(n - 1) .
    }
}

principalis {
    integer num : 5 .
    integer risultato .

    risultato : fattoriale(num) .

    //OUTPUT
    scribe_char('R') .
    scribe_char('E') .
    scribe_char('S') .
    scribe_char('=') .

    // Stampa il risultato (120)
    scribe_int(risultato) .
}
```