Manuale del Linguaggio Chiron (.chy)

Progetto Chiron

2025-05-22

# 1. Introduzione

## 1.1 Filosofia del linguaggio

Chiron nasce come ponte tra la semplicità espressiva di Python e la potenza e struttura dei linguaggi della famiglia C, in particolare C++. L’obiettivo principale è fornire un linguaggio semplice, chiaro e accessibile che introduca concetti come la tipizzazione statica, la gestione esplicita dello scope e la dichiarazione formale delle funzioni e classi, senza però sacrificare la leggibilità o la curva di apprendimento.

La filosofia di Chiron si fonda su tre principi cardine:

1. **Chiarezza esplicita**: ogni operazione sintattica è progettata per essere inequivocabile. Ad esempio, l’uso del simbolo : per indicare la direzione semantica di un’operazione (come ++ : i per il pre-incremento) evita ambiguità comuni nei linguaggi C-like.
2. **Accessibilità graduale**: pur introducendo concetti avanzati, Chiron è pensato per essere uno strumento didattico e di transizione. Programmatori provenienti da Python possono apprendere costrutti tipici dei linguaggi compilati senza trovarsi subito sommersi da complessità inutili.
3. **Coerenza strutturale**: la sintassi segue regole precise e omogenee, evitando eccezioni e casi speciali che generano confusione. Blocchi delimitati da parentesi graffe, terminazione obbligatoria con punto e virgola, commenti e dichiarazioni tipate formano un ecosistema coerente e prevedibile.

Chiron non ambisce, almeno inizialmente, a sostituire altri linguaggi nei contesti di produzione, ma vuole fornire un ambiente stabile e didattico per imparare a pensare in maniera strutturata e rigorosa. È un linguaggio nato per formare, per accompagnare e per evolversi insieme a chi lo utilizza.

## 1.2 Obiettivi e pubblico target

Chiron è stato ideato come un linguaggio educativo, di transizione e sperimentazione, con una sintassi rigorosa ma accessibile. I suoi obiettivi principali sono:

1. **Favorire l’apprendimento dei paradigmi strutturati**: Chiron permette di apprendere concetti fondamentali della programmazione tipata e compilata, come il controllo esplicito del flusso, la dichiarazione dei tipi, l’organizzazione in moduli e la gestione della memoria a livello concettuale.
2. **Semplificare la lettura e la manutenzione del codice**: la chiarezza sintattica e l’esplicitazione semantica riducono la possibilità di errori e favoriscono lo sviluppo collaborativo, anche in ambienti didattici.
3. **Offrire un linguaggio ponte**: progettato per chi conosce linguaggi dinamici come Python ma vuole avvicinarsi a linguaggi più formali come C++ o Rust, Chiron funge da passaggio intermedio per acquisire familiarità con costrutti più rigidi ma potenti.
4. **Essere un laboratorio concettuale**: grazie a un design aperto e flessibile, Chiron è pensato anche come base per la sperimentazione didattica e la progettazione di nuovi paradigmi linguistici. È quindi ideale per corsi universitari, workshop e progetti di ricerca.

Il pubblico target di Chiron include:

* **Studenti e autodidatti** che vogliono apprendere concetti avanzati in un ambiente più controllato e leggibile rispetto a C/C++.
* **Docenti e formatori** che necessitano di un linguaggio didattico coerente, senza eccezioni e con una sintassi ben documentata.
* **Sviluppatori Python** interessati a comprendere meglio il mondo della tipizzazione statica e del controllo esplicito dello scope.
* **Ricercatori e progettisti di linguaggi** che vogliono estendere o personalizzare un linguaggio esistente per prototipazione rapida.

Chiron non è progettato per essere il linguaggio "definitivo", ma un compagno di viaggio nella crescita del programmatore. Un ponte che unisce ciò che è noto con ciò che è ancora da esplorare.

## 1.3 Confronto con altri linguaggi (Python, C++)

Per comprendere appieno lo spirito di Chiron è utile confrontarlo con due dei linguaggi che ne hanno ispirato la progettazione: Python e C++.

| Caratteristica | Python | C++ | Chiron |
| --- | --- | --- | --- |
| Tipizzazione | Dinamica, implicita | Statica, esplicita | Statica, esplicita ma leggibile |
| Sintassi dei blocchi | Basata sull’indentazione | Basata su {} con terminazione a ; | Basata su {} con terminazione a ; |
| Obbligo di dichiarazione tipo | No | Sì | Sì |
| Paradigmi supportati | Imperativo, OOP, funzionale | Imperativo, OOP, generico, funzionale | Imperativo, OOP, in evoluzione |
| Gestione della memoria | Garbage collector implicito | Manuale (o smart pointers) | Concettualmente manuale (simulata) |
| Curva di apprendimento | Molto bassa | Alta | Graduale |
| Simbolismo semantico (:) | Non presente | Assente o sovraccarico semantico | Presente per esplicitare direzione |
| Commenti | # | //, /\* \*/ | #, //, .// |

Python è spesso considerato il linguaggio ideale per chi inizia: semplice, immediato, permissivo. Tuttavia, questa semplicità comporta una certa ambiguità strutturale, che può creare difficoltà nel passaggio a linguaggi più rigidi. C++, al contrario, è potente e flessibile, ma notoriamente complesso, con una sintassi densa e spesso criptica per i principianti.

Chiron si colloca nel mezzo: introduce la disciplina della tipizzazione statica e della dichiarazione esplicita, mantenendo però una sintassi leggibile, coerente e priva di costrutti oscuri. Il simbolo : per la direzionalità semantica, ad esempio, è un tentativo di rendere **esplicito ciò che in altri linguaggi è solo implicito o posizionale**.

Questo approccio lo rende adatto come linguaggio **ponte** per studenti, formatori e sviluppatori che vogliono crescere concettualmente senza dover affrontare subito tutta la complessità di C++ o Rust. )

## 1.4 Esempio di codice introduttivo

Di seguito un semplice programma scritto in Chiron che mostra i principali elementi sintattici del linguaggio: dichiarazione di variabili tipate, struttura dei blocchi, controllo di flusso, funzioni e utilizzo del simbolo : per operazioni direzionali.

# Questo programma calcola il fattoriale di un numero  
  
callable int factorial(int n) {  
 if (n <= 1) {  
 return 1;  
 } else {  
 return n \* factorial(n - 1);  
 };  
};  
  
callable void main() {  
 int x = 5;  
 int result = factorial(x);  
  
 print("Il fattoriale di " + x + " è " + result);  
};

Analisi degli elementi utilizzati:

* callable: parola chiave per la dichiarazione di funzioni.
* Tipizzazione esplicita: le variabili e i parametri richiedono un tipo (es. int).
* Blocchi delimitati da {} e istruzioni concluse da ; per coerenza e chiarezza.
* if e else con struttura chiara e obbligo di parentesi graffe.
* print(…​): funzione di output standard.
* Commenti singola linea con #, utilizzabili ovunque all’interno del codice.
* Concatenazione stringhe tramite +, coerente con altri linguaggi ad alto livello.
* Nessuna istruzione implicita: ogni azione deve essere espressa in modo esplicito.

Il programma segue uno stile fortemente leggibile, ispirato alla chiarezza di Python ma con la struttura e la disciplina tipica di C++. Il costrutto callable void main() rappresenta il punto di ingresso di ogni programma Chiron.

# 2. Sintassi di base

## 2.1 Regole generali di sintassi

Il linguaggio Chiron adotta una sintassi chiara, rigorosa e coerente. Le seguenti regole generali si applicano a tutte le strutture del linguaggio:

* **Tipizzazione esplicita**: ogni variabile, parametro o valore di ritorno deve essere associato a un tipo definito. Non è prevista inferenza automatica.
* **Dichiarazioni obbligatorie**: tutte le variabili devono essere dichiarate prima dell’uso. Non è consentita la creazione implicita.
* **Delimitatori di blocco**: ogni blocco di codice (funzione, condizione, ciclo, classe) è racchiuso tra parentesi graffe {}. Non esiste indentazione semantica obbligatoria, ma è fortemente raccomandata per la leggibilità.
* **Terminazione delle istruzioni**: ogni istruzione deve terminare con un punto e virgola ;. Anche le istruzioni singole all’interno di blocchi condizionali devono seguire questa regola.
* **Commenti**:
  + # commenta una singola riga (stile Python).
  + // apre un commento multilinea.
  + .// chiude un commento multilinea.
  + Il contenuto tra `//` e `.//` è ignorato dall’interprete. I commenti multilinea possono estendersi su più righe. Non è permesso annidare più commenti multilinea.
* **Case sensitivity**: Chiron distingue tra maiuscole e minuscole nei nomi di variabili, funzioni, classi e tipi. Le parole chiave sono tutte in minuscolo.
* **Nomi validi**:
  + Devono iniziare con una lettera (a-z, A-Z) o con il simbolo \_.
  + Possono contenere lettere, numeri e \_, ma non simboli speciali.
  + Non possono coincidere con parole chiave riservate.
* **Struttura dei file**:
  + I file sorgente devono avere estensione .chy.
  + Ogni file può contenere più funzioni, dichiarazioni e classi, ma solo una funzione main() sarà considerata punto di ingresso, se presente.
* **Spaziature e linee vuote**: non influiscono sulla semantica del codice, ma è buona pratica usarle per separare logicamente blocchi e migliorare la leggibilità.

Queste regole costituiscono la base comune per la scrittura di codice Chiron valido. Le sezioni successive ne dettaglieranno l’applicazione nei vari costrutti sintattici.

## 2.2 Blocchi e indentazione

## 2.3 Commenti (#, .//, //)

## 2.4 Terminazione delle istruzioni

## 2.5 Simbolo di direzione :

Il simbolo : viene utilizzato per rendere esplicita la direzione semantica dell’operazione. Ad esempio:

* Pre-incremento: ++ : i → incrementa prima di utilizzare
* Post-incremento: i : ++ → utilizza prima di incrementare

Questa notazione migliora la chiarezza semantica e riduce ambiguità sintattiche rispetto ad altri linguaggi.

# 3. Tipi di dati

## 3.1 Tipi primitivi

## 3.2 Tipi complessi

## 3.3 Dichiarazione e inizializzazione

## 3.4 Conversione tra tipi

# 4. Espressioni e operatori

## 4.1 Operatori aritmetici

## 4.2 Operatori logici e relazionali

## 4.3 Operatori di assegnazione

## 4.4 Precedenza e associatività

# 5. Controllo di flusso

## 5.1 Condizionali: if, else if, else

## 5.2 Cicli: while, for

## 5.3 Interruzioni di flusso: break, continue

# 6. Funzioni

## 6.1 Dichiarazione e sintassi base (callable)

## 6.2 Tipi di ritorno

## 6.3 Parametri opzionali e default

## 6.4 Funzioni come variabili

# 7. Classi e oggetti

## 7.1 Dichiarazione di classi

## 7.2 Attributi, metodi e costruttore

## 7.3 this e visibilità interna

## 7.4 Ereditarietà e overloading

# 8. Scope e visibilità

## 8.1 Regole di visibilità (global, static, const)

## 8.2 Scope locale e globale

## 8.3 Shadowing e gestione dei conflitti

# 9. Gestione delle eccezioni

## 9.1 Sintassi: try, except, finally

## 9.2 Tipi di eccezioni

## 9.3 Generazione di errori (raise)

# 10. Moduli e librerie standard

## 10.1 Struttura dei file .chy

## 10.2 Importazione dei moduli

## 10.3 Libreria standard prevista

# 11. Input/Output

## 11.1 Funzioni di I/O standard: print, input

## 11.2 Gestione dei file: lettura, scrittura, apertura

# 12. Ambiente di esecuzione

## 12.1 Esecuzione di uno script .chy

## 12.2 Prompt interattivo >

## 12.3 Comandi speciali (es. .exit, .help)

# 13. Estensioni future

## 13.1 Lambda e funzioni anonime

## 13.2 Meta-programmazione

## 13.3 Supporto a modelli funzionali o concorrenti