Manuale del Linguaggio Chiron (.chy)

Progetto Chiron

Version 0.2, 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 factorial(int n) -> int {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    };
};

callable main() -> void {
    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. Esso è però opzionale: nel caso di assenza dell'entry point 'main()' l'interprete inizierà ad eseguire il codice *'globale'*, come in python.

In caso la funzione 'main()' sia però presente, tutto quanto il codice 'globale' sarà ignorato.

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

Chiron utilizza le parentesi graffe `{}` per delimitare blocchi di codice. Questo approccio, tipico dei linguaggi della famiglia C, garantisce una maggiore chiarezza e riduce il rischio di errori causati da indentazione errata.

L'indentazione è **opzionale** e serve solo a migliorare la leggibilità. Non influenza in alcun modo l'esecuzione del codice.

Esempio:

```
if (x > 0) {
    print("Positivo");
} else {
    print("Negativo o zero");
}
```

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

Chiron supporta tre tipi di commento:

- Commenti monolinea: iniziano con `#` e terminano a fine riga.
- Commenti multilinea: iniziano con `//` e terminano con `.//`. Possono estendersi su più righe.

Esempi:

```
# Questo è un commento su una riga

// Questo è un commento
   su più righe
   e termina qui.//
```

2.4 Terminazione delle istruzioni

Ogni istruzione in Chiron deve terminare con un punto e virgola `;`, come nei linguaggi C-like. Questo consente una sintassi prevedibile e chiara, e semplifica l'analisi del codice.

Esempio:

```
int x = 10;
x = x + 1;
```

2.5 Simbolo di direzione `:`

Il simbolo `:` viene utilizzato per rendere esplicita la **direzione semantica** di un'operazione. Questo approccio evita ambiguità comuni in altri linguaggi.

Esempi di utilizzo:

- Pre-incremento: `++ : i` → incrementa `i`, poi restituisce il valore incrementato.
- Post-incremento: `i : ++` → restituisce `i`, poi lo incrementa.
- Accesso a funzione: `obj : metodo()` → chiama `metodo()` su `obj`.

Questo meccanismo rende le operazioni più leggibili e gestibili per l'interprete.

3. Tipi di dati

3.1 Tipi primitivi

Chiron supporta i seguenti tipi primitivi:

- `int`: intero (es. `42`)
- `float`: numero in virgola mobile (es. `3.14`)
- `bool`: booleano (`true`, `false`)
- `char`: singolo carattere (`'a'`, `'%'`)
- `str`: sequenza di caratteri (`"ciao"`)
- `callable`: una funzione

```
int x = 10;
float pi = 3.14;
bool attivo = true;
char iniziale = 'A';
str saluto = "Ciao mondo";
```

3.2 Tipi complessi

Tipi complessi predefiniti:

- `array<T>`: vettore di elementi del tipo `T`
- `tuple<T1, T2, ···>`: tupla con tipi misti
- `map<K, V>`: dizionario con chiavi di tipo `K` e valori di tipo `V`

```
array<int> numeri = [1, 2, 3];
tuple<str, int> persona = ("Luca", 30);
map<str, int> età = {"Luca": 30, "Anna": 25};
```

3.3 Dichiarazione e inizializzazione

Le variabili devono essere dichiarate esplicitamente con il loro tipo. L'inizializzazione può avvenire al momento della dichiarazione o in un secondo momento.

```
int a;
a = 5;

float b = 2.5;
```

3.4 Conversione tra tipi

Chiron supporta la conversione esplicita tra tipi compatibili tramite la sintassi:

```
float x = 3;
int y = (int) x;  # Conversione esplicita
```

Alcune conversioni implicite sono consentite (es. `int` → `float`), ma le conversioni che potrebbero comportare perdita di informazione devono essere esplicitate.

4. Espressioni e operatori

4.1 Operatori aritmetici

- `+` somma
- `-` sottrazione
- `*` moltiplicazione
- `/` divisione
- `%` modulo

```
int a = 5 + 3;
float b = 10.0 / 4;
```

4.2 Operatori logici e relazionali

- `==` uguaglianza
- `!=` disuguaglianza
- `<`, `⇐`, `>`, `>=` confronti
- `and` AND logico
- `or` OR logico
- `not` NOT logico

```
if (a > 0 and b != 0) {
    print("Valori validi");
}
```

4.3 Operatori di assegnazione

- `=` assegnazione semplice
- `+=`, `-=`, `*=`, `/=`, `%=` assegnazioni combinate

```
auto x = 10;
auto x += 5;  # x ora vale 15
```

4.4 Precedenza e associatività

Chiron segue la precedenza classica dei linguaggi C-like. Gli operatori tra parentesi hanno la precedenza più alta. È sempre consigliato l'uso esplicito delle parentesi per evitare ambiguità.

Ordine di precedenza (dal più alto al più basso):

1. `( )` (parentesi)
2. `++`, `--` (*consigliato* l'uso con ':')
3. `*`, `/`, `%`
4. `+`, `-`
5. Relazionali (`<`, `>`, `<=`, `>=`)
6. Uguaglianza (`==`, `!=`)
7. `&&`
8. `||`
9. `=`, `+=`, ecc.

```
int risultato = (a + b) * c;
```

5. Controllo di flusso

5.1 Condizionali: `if`, `else if`, `else`

Chiron utilizza la classica struttura condizionale, simile ai linguaggi C-like. Ogni blocco deve essere delimitato da parentesi graffe `{}`. Le condizioni devono essere esplicitamente booleane.

```
if (x > 0) {
    print("Positivo");
} else if (x == 0) {
    print("Zero");
} else {
    print("Negativo");
}
```

5.2 Cicli: `while`, `for`

Ciclo `while`

Il ciclo `while` ripete il blocco finché la condizione è vera.

```
int i = 0;
while (i < 5) {
    print(i);
    i : ++;
}
```

Ciclo `for`

Il ciclo `for` segue la struttura classica con dichiarazione, condizione e incremento.

```
for (int i = 0; i < 5; i : ++) {
    print(i);
}
```

5.3 Interruzioni di flusso: `break`, `continue`

- `break` termina immediatamente il ciclo più interno.
- `continue` salta all'iterazione successiva del ciclo.

```
for (int i = 0; i < 10; i : ++) {
    if (i == 5) {
        continue;  # Salta il 5
```

```
    }
    if (i == 8) {
        break;     # Interrompe il ciclo a 8
    }
    print(i);
}
```

6. Funzioni

6.1 Dichiarazione e sintassi base (`callable`)

Le funzioni in Chiron si dichiarano usando il costrutto `callable`, seguito dal nome, dai parametri tipizzati e dal tipo di ritorno. Il corpo della funzione è racchiuso tra parentesi graffe.

```
callable somma(int a, int b) -> int {
    return a + b;
}
```

Le funzioni possono essere dichiarate a livello globale o all'interno di classi.

6.2 Tipi di ritorno

Il tipo di ritorno deve essere specificato esplicitamente dopo il simbolo `->`. Per funzioni che non restituiscono valori, si utilizza `void`.

```
callable stampaMessaggio() -> void {
    print("Benvenuto in Chiron!");
}
```

Una funzione può restituire qualunque tipo, anche strutture complesse o classi.

6.3 Parametri opzionali e default

Chiron supporta parametri opzionali, definiti con un valore di default.

```
callable saluta(str nome = "Utente") -> void {
    print("Ciao, " + nome + "!");
}
```

Questa funzione può essere invocata con o senza argomenti.

```
saluta();          # Output: Ciao, Utente!
saluta("Alice");   # Output: Ciao, Alice!
```

6.4 Funzioni come variabili

Il costrutto `callable` visto in precedenza non è li a caso: infatti in chiron le funzioni non sono nient'altro che variabili. In particolare le funzioni vengono salvate come stringhe multi-linea costanti (`const str`)

Di conseguenza concetti come *puntatori a funzione* non esistono in quanto concettualmente si sta lavorando con stringhe

Assegnazione di funzione

```
callable moltiplica(int a, int b) -> int {
    return a * b;
}

callable operazione = moltiplica;
print( operazione(3, 4) );  # Output: 12
```

Questa caratteristica abilita l'uso di funzioni di ordine superiore e apre la strada alla programmazione funzionale.

7. Classi e oggetti

7.1 Dichiarazione di classi

Le classi in Chiron sono definite con la parola chiave `class`, seguita dal nome della classe. Il corpo della classe è racchiuso tra parentesi graffe e può contenere attributi, metodi e costruttori.

```
class Punto {
    int x;
    int y;

    callable Punto(int x, int y) -> void {
        this.x = x;
        this.y = y;
    }
}
```

7.2 Attributi, metodi e costruttore

Gli attributi sono variabili tipizzate dichiarate direttamente all'interno della classe. I metodi sono funzioni che operano sull'istanza corrente. Il costruttore ha lo stesso nome della classe e non ha tipo di ritorno.

```
class Rettangolo {
    int base;
    int altezza;

    callable Rettangolo(int b, int h) -> void {
        this.base = b;
        this.altezza = h;
    }

    callable area() -> int {
        return base * altezza;
    }
}
```

7.3 `this` e visibilità interna

La parola chiave `this` è utilizzata per riferirsi all'istanza corrente dell'oggetto, ed è obbligatoria quando c'è ambiguità tra parametri e attributi.

```
this.base = base;  # disambiguazione tra parametro e attributo
```

In futuro potranno essere introdotti modificatori di visibilità (`private`, `protected`, `public`), ma attualmente tutte le proprietà sono accessibili.

7.4 Ereditarietà e overloading

Chiron supporta l'ereditarietà semplice tramite la sintassi `: nomeClassePadre`.

```
class Figura {
    callable tipo() -> str {
        return "Figura generica";
    }
}

class Cerchio : Figura {
    callable tipo() -> str {
        return "Cerchio";
    }
}
```

Il metodo `tipo()` è ridefinito nella sottoclasse (`overriding`). Chiron consente anche l'overloading, cioè la definizione di più metodi con lo stesso nome ma parametri diversi.

```
class Operazioni {
    callable somma(int a, int b) -> int {
        return a + b;
    }

    callable somma(float a, float b) -> float {
        return a + b;
    }
}
```

L'overloading si basa sulle firme delle funzioni (tipi e numero di parametri).

8. Scope e visibilità

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

In Chiron, la visibilità e il comportamento delle variabili possono essere controllati con modificatori espliciti:

- `global`: indica che la variabile è definita nello scope globale ed è accessibile da qualsiasi punto del programma.
- `static`: la variabile o funzione mantiene il suo valore tra le invocazioni e non viene ricreata ad ogni esecuzione.
- `const`: definisce una costante il cui valore non può essere modificato dopo l'inizializzazione.

```
global int MAX_UTENTI = 100;
const float PI = 3.1416;
static int contatore = 0;
```

8.2 Scope locale e globale

Chiron distingue in modo chiaro tra variabili locali (dichiarate all'interno di funzioni o blocchi) e globali (dichiarate all'esterno). Le variabili locali nascondono quelle globali con lo stesso nome.

```
int x = 10;

callable esempio() -> void {
    int x = 5;       # nasconde la variabile globale
    print(x);        # stampa: 5
}
```

Per accedere alla variabile globale `x` in questo contesto, si usa la parola chiave `global`.

```
callable esempio2() -> void {
    global x;
    print(x);        # stampa: 10
}
```

8.3 Shadowing e gestione dei conflitti

Chiron permette lo shadowing, ma emette un warning in fase di compilazione se una variabile locale ombreggia un'identica variabile globale. È consigliato evitare nomi duplicati per chiarezza semantica.

```
int valore = 50;
```

```
callable stampa() -> void {
    int valore = 20;  # warning: 'valore' ombreggia variabile globale
    print(valore);    # stampa: 20
}
```

9. Gestione delle eccezioni

9.1 Sintassi: `try`, `except`, `finally`

Chiron fornisce un meccanismo robusto per la gestione delle eccezioni. Il blocco `try` racchiude il codice potenzialmente fallibile, seguito da uno o più blocchi `except`, e un opzionale blocco `finally`.

```
try {
    int x = div(10, 0);
}
except ZeroDivisionError as e {
    print("Errore: divisione per zero");
}
finally {
    print("Operazione terminata");
}
```

9.2 Tipi di eccezioni

Le eccezioni sono oggetti di tipo specifico. Chiron prevede eccezioni di base come:

- `ZeroDivisionError`
- `ValueError`
- `IndexError`
- `FileNotFoundError`
- `GenericError` (per casi generici)

Le eccezioni possono essere personalizzate creando classi che ereditano da `Exception`.

```
class ErroreLogin : Exception {
    str motivo;

    callable ErroreLogin(str m) -> void {
        this.motivo = m;
    }
}
```

9.3 Generazione di errori (`raise`)

Per sollevare un'eccezione si utilizza la parola chiave `raise`.

```
static callable div(int a, int b) -> int {
    if b == 0 {
```

```
        raise ZeroDivisionError("Divisione per zero");
    }
    return a / b;
}
```

Le eccezioni non gestite causano la terminazione del programma, mostrando un traceback sintetico.

10. Moduli e librerie standard

10.1 Struttura dei file `.chy`

I programmi Chiron sono scritti in file con estensione `.chy`. Ogni file può contenere dichiarazioni di funzioni, classi e costanti, ed è trattato come un modulo.

La struttura tipica di un file `.chy`:

```
# Questo è un file chiamato 'util.chy'

static callable somma(int a, int b) -> int {
    return a + b;
}

const int VERSIONE = 1;
```

I file `.chy` possono essere importati in altri script, rendendo disponibili le entità dichiarate.

10.2 Importazione dei moduli

Per importare un modulo, si usa la parola chiave `import`. Il nome del file (senza estensione `.chy`) è usato come nome del modulo.

```
import util;

int risultato = util : somma(4, 5);
print(risultato);  # stampa: 9
```

È possibile rinominare un modulo al momento dell'importazione con `as`.

```
import util as u;

print(u : VERSIONE);  # stampa: 1
```

L'importazione avviene in fase di esecuzione e si basa sul percorso relativo del file. I moduli devono trovarsi nella stessa directory o in una delle directory specificate nel path di Chiron.

10.3 Libreria standard prevista

La libreria standard di Chiron fornisce moduli integrati che facilitano operazioni comuni. I moduli più importanti includono:

- `std.io` – Input/output (es. `print`, `input`)

- `std.math` – Funzioni matematiche comuni (`abs`, `sqrt`, `pow`)
- `std.fs` – Gestione file (lettura, scrittura) e formati comuni come JSON, SQL etc.
- `std.str` – Manipolazione di stringhe
- `std.time` – Gestione del tempo e delle date
- `std.sys` – Informazioni e comandi di sistema

Esempio di utilizzo di una funzione dalla libreria standard:

```
import std.math;

float radice = std.math : sqrt(16.0);
print(radice);  # stampa: 4.0
```

La libreria standard di Chiron è progettata per essere minimale ma estendibile. Nuovi moduli possono essere installati e importati come qualsiasi altro file `.chy`.

11. Input/Output

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

Chiron fornisce funzioni integrate per l'interazione con l'utente tramite console.

`print(…)`

Stampa uno o più valori sulla console, separati da uno spazio.

```
print("Hello, World!");
print("Valore:", 42, true);
```

Supporta tipi primitivi e stringhe, e converte automaticamente i tipi in formato testuale.

`input(prompt: str) → str`

Legge una riga da input utente e la restituisce come stringa.

```
str nome = input("Inserisci il tuo nome: ");
print("Benvenuto", nome);
```

11.2 Gestione dei file: lettura, scrittura, apertura

Chiron supporta la gestione dei file tramite l'interfaccia `file`, presente nel modulo `std.fs`.

```
import std.fs;

file f = std.fs : open("dati.txt", "w");
f : write("Linea 1\n");
f : close();
```

Modalità di apertura:

- `"r"` – Lettura (il file deve esistere)
- `"w"` – Scrittura (sovrascrive se esiste)
- `"a"` – Aggiunta (scrive in fondo al file)
- `"rw"` – Lettura e scrittura

Per ognuno di questi sistemi esiste le varianti più comuni come 'b' per lettura di tipo bytes (esempio 'rb' per lettura dei bytes si un file)

Metodi disponibili su oggetti `file`:

- `write(str data)` – Scrive nel file
- `read() → str` – Legge l'intero contenuto
- `readline() → str` – Legge una singola linea
- `close()` – Chiude il file

Esempio di lettura:

```
file f = std.fs : open("dati.txt", "r");
str contenuto = f : read();
print(contenuto);
f : close();
```

12. Ambiente di esecuzione

12.1 Esecuzione di uno script `.chy`

Gli script `.chy` sono eseguiti da un interprete Chiron. Si possono eseguire dalla riga di comando:

```
$ chiron mio_script.chy
```

È possibile passare argomenti al programma tramite `args`, una lista disponibile nel contesto globale.

```
print("Numero di argomenti:", args : size());
```

12.2 Prompt interattivo `>`

Chiron include un REPL (Read-Eval-Print Loop), accessibile semplicemente eseguendo:

```
$ chiron
```

Nel REPL, ogni riga viene interpretata ed eseguita immediatamente.

```
> int x = 10;
> x : ++;
> print(x);
11
```

Il prompt supporta comandi speciali e sintassi multilinea.

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

Nel prompt interattivo sono disponibili comandi speciali:

- `.exit` – Chiude il REPL
- `.help` – Mostra la guida in linea
- `.clear` – Pulisce la schermata
- `.env` – Mostra le variabili definite

```
> .help
Comandi disponibili:
  .exit    → Esce dal prompt
  .help    → Mostra questo messaggio
  .env     → Elenca variabili e moduli caricati
```

Il REPL è pensato per esperimenti rapidi, test e apprendimento interattivo.

13. Estensioni future

13.1 Lambda e funzioni anonime

Chiron prevede di integrare nel prossimo futuro il supporto a funzioni anonime (lambda), per migliorare la concisione e la flessibilità del codice. Le lambda permetteranno di definire funzioni inline, senza dover creare dichiarazioni formali, facilitando la programmazione funzionale e la scrittura di callback o funzioni di ordine superiore.

Sintassi proposta:

`lambda (parametri) : espressione`

Esempio

```
auto somma = lambda (a, b) : a + b;
```

13.2 Meta-programmazione

Una delle direzioni evolutive di Chiron riguarda la meta-programmazione, cioè la possibilità di scrivere codice che genera o manipola altro codice a tempo di compilazione o esecuzione.

Questo permetterà di introdurre macro più potenti e generiche, sistemi di riflessione limitati e template evoluti, con l'obiettivo di mantenere però la semplicità e la chiarezza del linguaggio.

13.3 Supporto a modelli funzionali o concorrenti

Chiron intende esplorare in futuro modelli di programmazione funzionale e concorrente, per adattarsi alle esigenze moderne.

In particolare, si valutano:

- Introduzione di tipi immutabili e funzioni pure
- Costrutti per la gestione della concorrenza (thread, async/await)
- Sistemi di sincronizzazione semplificati
- Meccanismi per la gestione della concorrenza basata su messaggi o attori

L'obiettivo è fornire strumenti efficaci per la scrittura di software moderno, mantenendo il bilanciamento con la semplicità e leggibilità che caratterizza Chiron.