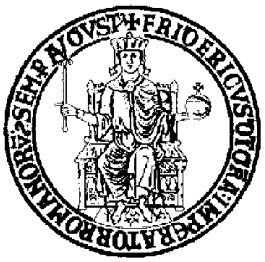
Università degli Studi di Napoli
Federico II



Dipartimento di Ingegneria Elettrica e delle Tecnologie dell'Informazione
Corso di Laurea Triennale in Informatica

Elaborato di Laurea

# *Design e Sviluppo del linguaggio di programmazione "Basalt"*

Relatore:
Ch.mo Prof. Faella Marco

Candidato:
De Rosa Francesco
Matr. N86004379

Anno Accademico
2024/2025

# Indice

# 1 Design del linguaggio

## 1.1 Introduzione

Basalt nasce con l'intenzione di offrire una alternativa moderna a linguaggi di programmazione consolidati come C e C++. Nonostante questi linguaggi siano ancora ben lontani dall'essere considerati obsoleti, è innegabile che comincino a mostrare i segni del tempo.

L'obiettivo di Basalt è quello di superare alcune delle rigidità e mancanze del linguaggio C, proponendosi come un linguaggio più flessibile, espressivo ed ergonomico, mirando ad equipaggiare i programmatori con strumenti di programmazione all'avanguardia, tipici dei linguaggi di più alto livello. Basalt offre infatti strumenti avanzati quali riflessione ed introspezione a tempo di compilazione, supporto alla programmazione generica e un framework di unit-testing integrato direttamente nel linguaggio.

Basalt, così come Go, Rust, Zig, Odin e molti altri linguaggi di basso livello di nuova concezione, non è un linguaggio orientato agli oggetti. La mancata adozione del paradigma della programmazione orientata agli oggetti in tutti questi linguaggi, e in particolare in Basalt, non comporta una riduzione dell'espressività e della modularità del codice. Anzi, ha consentito scelte innovative di language design che hanno portato a soluzioni più moderne ed eleganti. Queste scelte hanno introdotto costrutti che risolvono gli stessi problemi della programmazione orientata agli oggetti, ma in modo più efficiente, mantenendo sia l'astrazione che il controllo a basso livello, migliorando così la flessibilità e la performance complessiva del codice.

Basalt si distingue come l'unico linguaggio che offre un supporto alla riflessione paragonabile in termini di completezza e funzionalità a linguaggi come Java, pur mantenendo un ruolo funzionalmente analogo a quello del C o del C++. Al contempo, Basalt adotta un'estetica minimale e orientata alla semplicità, in linea con l'approccio stilistico di Go. Pertanto, questi quattro linguaggi costituiranno i principali riferimenti per il confronto nel prosieguo della discussione.

### 1.1.1 Confronto con altri linguaggi

Di seguito è stata presentata una tabella comparativa che mette a confronto Basalt con C, C++, Go e Java, evidenziando le caratteristiche comuni tra questi linguaggi e Basalt.

Si tenga presente che per molte delle feature elencate, sarebbe stato possibile considerare i loro corrispettivi inversi. Ad esempio, per la feature "gestione manuale della memoria", si sarebbe potuto considerare anche la feature "gestione automatica della memoria". L'obbiettivo di questa tabella comparativa è dunque non quello di dipingere Basalt come un linguaggio provvisto di ogni feature, ma semplicemente di cosndierare ogni feature di Basalt e verificare in quale dei linguaggi scelti come termine di paragone essa sia presente.

| ***Features*** | ***Basalt*** | ***C*** | ***C++*** | ***Go*** | ***Java*** |
|---|---|---|---|---|---|
| gestione manuale della memoria | ✓ | ✓ | ✓ | | |
| programmazione generica | ✓ | | ✓ | ✓ | ✓ |
| union/variant | ✓ | ✓ | ✓ | | ✓ |
| introspezione e riflessione | ✓ | | | ✓ | |
| zero-cost-abstractions | ✓ | ✓ | ✓ | | |
| compilato nativamente in linguaggio macchina | ✓ | ✓ | ✓ | ✓ | |
| non richiede header-files ed è invece basato su package/moduli | ✓ | | | ✓ | ✓ |
| non richiede che la definizione di un simbolo ne preceda l'utilizzo | ✓ | | | ✓ | ✓ |
| overloading di funzioni/metodi | ✓ | | ✓ | ✓ | ✓ |
| metaprogrammazione con supporto per annotazioni e decorator | ✓ | | | | ✓ |
| assenza di allocazioni di memoria nascoste o implicite | ✓ | ✓ | ✓ | | |
| framework di unit-testing integrato nel linguaggio | ✓ | | | ✓ | |
| ***Score:*** | 12/12 | 5/12 | 7/12 | 7/12 | 6/12 |

Tabella 1: Confronto tra Basalt e altri linguaggi di programmazione

### 1.1.2 Struttura di un programma Basalt

Come precedentemente menzionato, Basalt si discosta dall'utilizzo di header files tipico di C e C++, optando invece per un sistema di gestione dei pacchetti simile a quello adottato da Java. In particolare, il sistema dei pacchetti di basalt prevede che all'intero di un file apparenente ad un dato pacchetto, sia visibile il contenuto pubblico e non di tutti gli altri file dello stesso pacchetto, e il contenuto pubblico dei package importati.

Ogni file sorgente contenente codice Basalt deve possedere una intestazione composta dalla dichiarazione del package corrente, ovvero il package a cui il file appartiene, e da una lista di package importati dal file, necessari al suo funzionamento.

Così come C, C++, Zig, Rust, Go, Jai, Odin e molti altri, il flusso di esecuzione parte da una chiamata fittizia ad una funzione speciale detta entry-point del programma. Così come da convenzione, tale funzione prende il nome di "main". Tale funzione deve necessariamente essere in un package di nome "main".

```
package main;
import console;

func main() {
    println("Hello, World!");
}
```

In maniera analoga a quanto è possibile vedere in Java, in Basalt importare un package non è una precondizione necessaria per l'utilizzo delle funzioni di tale package. É infatti possibile utilizzare la funzione `println` anche senza importare il package `console`, semplicemente riferendosi a tale funzione con il suo nome completo:

```
package main;

func main() {
    console::println("Hello, World!");
}
```

### 1.1.3 Indipendenza dall'ordine di definizione

Così come in Java, Rust e Go, e a differenza di C e C++, Basalt prevede che ogni definizione possa essere spostata in qualunque punto di un file sorgente o addirittura migrata in un altro file sorgente dello stesso package senza compromettere la correttezza del programma. In altri termini, in Basalt ogni definizione è accessibile non solo dalle definzioni che la succedono ma anche da quelle che la precedono.

L'indipendenza dall'ordine di definizione in un linguaggio di programmazione semplifica notevolmente il refactoring e l'utilizzo del codice. Il programmatore può riorganizzare e ottimizzare il codice senza dover preoccuparsi di errori di compilazione dovuti a riferimenti non ancora definiti. Questo favorisce una maggiore modularità e facilita il mantenimento del codice, poiché le modifiche possono essere apportate in modo più flessibile e incrementale. Inoltre, consente di migliorare la leggibilità del codice, organizzandolo in modo logico piuttosto che cronologico.

In un contesto di sviluppo collaborativo, questa caratteristica è particolarmente vantaggiosa, poiché diversi sviluppatori possono lavorare su parti diverse del codice senza doversi coordinare strettamente sull'ordine delle definizioni. Ciò riduce i conflitti di merge e accelera il processo di sviluppo. Anche l'aggiunta di nuove funzionalità o la correzione di bug risulta più semplice, in quanto le nuove definizioni possono essere inserite esattamente dove hanno più senso logico, senza dover riscrivere o spostare altre parti del codice esistente.

Ciò significa che il seguente codice è valido. Il compilatore è capace di risolvere correttamente il riferimento alla funzione `sum` anche se essa è definita dopo il suo primo utilizzo (ovvero la chiamata avvenuta nella funzione `main`).

```
package main;

func main() {
    var result : Int = sum(3, 5);
    console::print("The sum of 3 and 5 is: ");
    console::println(result);
}

func sum(a: Int, b: Int) -> Int {
    return a + b;
}
```

## 1.2 Variabili e costanti

Variabili e costanti sono dei contenitori logici capaci di contenere dei valori decisi a tempo di esecuzione. Ci si potrà riferire al valore contenuto in un dato istante di tempo da una variabile o da una costante utilizzando il suo nome. Tramite un apposito costrutto detto assegnazione, è possibile riassegnare il valore di una variabile, ciò non è però possibile per una costante, la quale una volta dichiarata non potrà mai cambiare valore. L'utilizzo delle costanti in Basalt è infatti concesso solo nei contesti dove il loro valore non viene modificato (accesso in sola lettura).

### 1.2.1 Dichiarazione di variabili

La dichiarazione di una variabile può avvenire con inizializzazione o senza, laddove un valore di inizializzazione sia mancante il valore di tale variabile sarà casuale. Ci si aspetta che in tale scenario un valore venga poi assegnato in un secondo momento. Qualunque sia la tipologia di dichiarazione scelta, essa deve essere introdotta dalla keyword var, seguida dal nome della variabile, dai due punti e dal tipo di tale variabile.

```
var x : Int = 6;
var y : Int;
```

### 1.2.2 Dichiarazione di costanti

In maniera del tutto analoga a quanto detto per le variabili, la dichiarazione delle costanti deve essere introdotta dalla keyword const, seguita dal nome della costante, dai due punti e dal tipo, solo che a differenza di una variabile, una costante deve necessariamente essere inizializzata in sede di dichiarazione.

```
const pi : Float = 3.14;
```

Le costanti possono eventualmente essere rimpiazzate dal loro valore in tutte le loro occorrenze qualora tale valore sia noto a tempo di compilazione, e qualora l'indirizzo di memoria della costante non sia utilizzato all'interno del codice sorgente in alcun modo.
É comunque possibile assegnare ad una costante (in sede di dichiarazione) un valore noto a tempo di esecuzione, in tal caso l'ottimizzazione appena descritta sarà impossibile, anche se potrebbe comunuqe essere rimpiazzato il suo utilizzo con l'espressione in sè qualora tale utilizzo sia unico, e qualora ciò non comporti un cambiamento osservabile del programma ipotizzando un'esecuzione single thread.

## 1.3 Controllo del flusso di esecuzione

Con il termine "Control-Flow", o in italiano controllo del flusso di esecuzione, si intende l'iniseme dei costrutti che rendono l'esecuzione del codice non lineare, in particolare in Basalt essi sono cicli iterativi e branch condizionali.

### 1.3.1 Branch condizionali

Un branch condizionale, anche chiamato "if-statement", è un costrutto che consente di eseguire una porzione di codice solo se una certa conzione booleana è vera. Ad esmpio si consideri il seguente frammento di codice che illustra l'utilizzo di un if-statement.

```
var x : Int = math::random(0,10);
if (x % 2 == 0) {
    console::println("x is even");
}
```

In questo codice, l'istruzione `console::println("x is even")` sarà eseguita solo nel caso in cui il valore numerico intero attualmente contenuto nella variabile x sarà pari.
É possibile aggiungere un blocco di codice da eseguire nel caso in cui la condizione sia falsa utilizzando la keyword else. Ad esempio è possibile stampare del testo che informi l'utente del fatto che la variabile x contiene un valore dispari.

```
var x : Int = math::random(0,10);
if (x % 2 == 0) {
    console::println("x is even");
}
else {
    console::println("x is odd");
}
```

In Basalt l'indentazione non è rilevante, per cui, se lo si preferisce, è accettato (anche se sconsigliato) disporre la keyword else sulla stessa riga della chiusura della parentesi graffa relativa al blocco di codice da eseguire nel caso in cui la condizione sia vera.

### 1.3.2 Ciclo while

Il ciclo while è un costrutto utilizzato per ripetere una certa porzione di codice finchè una certa condizione booleana rimane vera. Il corpo del ciclo viene eseguito solo dopo aver controllato la condizione booleana. Si consideri ad esempio il seguente frammento di codice dove è presentato un ciclo while a scopo esemplificativo:

```
var i : Int = 0;
while (i < 10) {
    console.println(x);
    i = i + 1;
}
```

L'esecuzione di tale ciclo comporta la stampa in console dei numeri da 0 a 9. Più in generale, si può dire che un ciclo while è composto da condizione e corpo, e che la sua esecuzione avviene secondo il seguente diagramma di flusso (flow-chart).

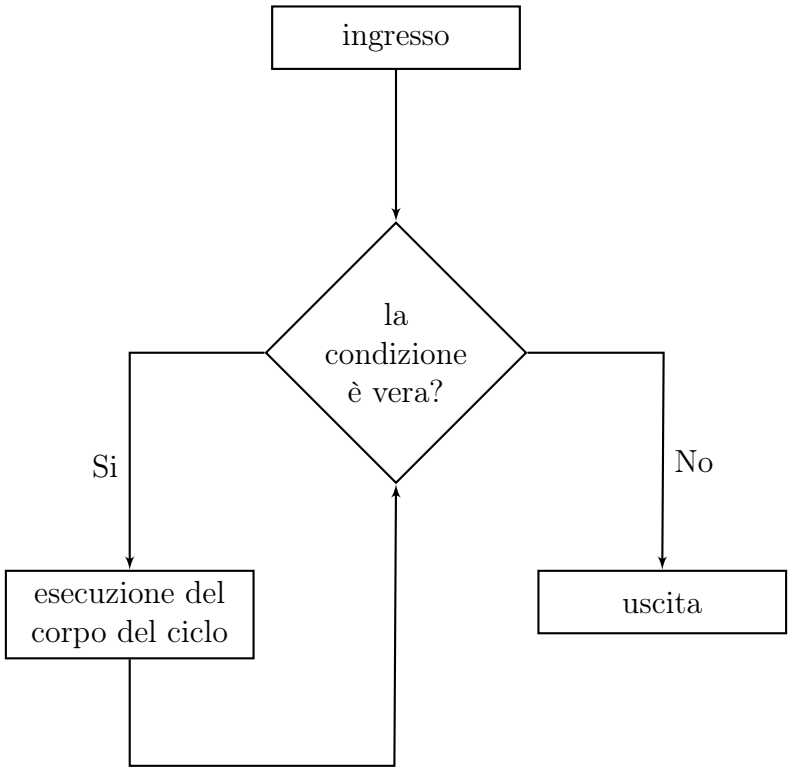



Figura 1: Diagramma di flusso del ciclo while

### 1.3.3 Ciclo until

Il ciclo until è un costrutto utilizzato per ripetere una certa porzione di codice finchè una certa condizione booleana rimane falsa. Il corpo del ciclo viene eseguito prima di aver controllato la condizione booleana. Si consideri ad esempio il seguente frammento di codice dove è presentato un ciclo while a scopo esemplificativo:

```
var i : Int = 0;
until (i > 10) {
    console.println(x);
    i = i + 1;
}
```

L'esecuzione di tale ciclo comporta la stampa in console dei numeri da 0 a 10. Così come per il ciclo while, si può dire che un ciclo until è composto da condizione e corpo, e che la sua esecuzione avviene secondo il seguente diagramma di flusso (flow-chart).

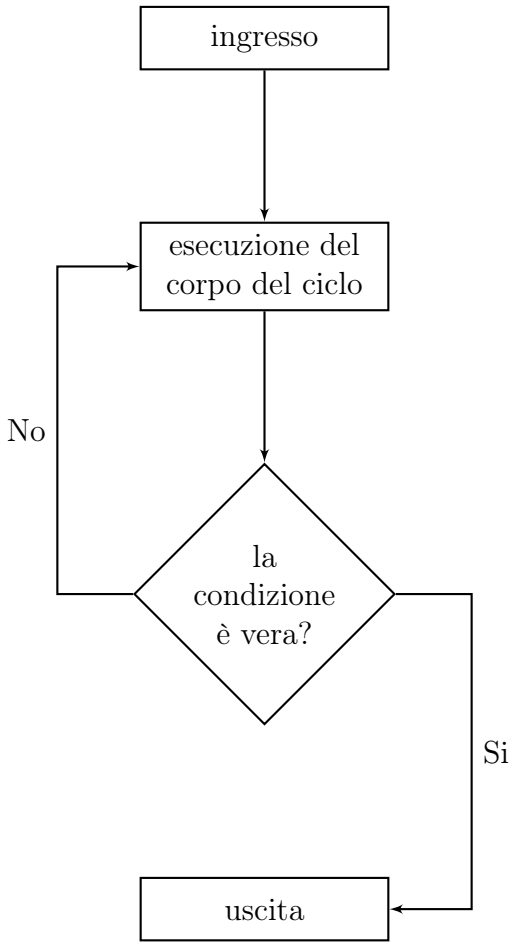



Figura 2: Diagramma del ciclo until

# 2 Architettura

In questo capitolo, sarà documentata l'architettura interna del compilatore. Per analizzare in dettaglio la suddetta architettura, saranno utilizzati svariate rappresentazioni grafiche in veste di diagramma, quali UML-Class-Diagram, UML-Sequence-Diagram, UML-Component-Diagram. Si tenga presente che tali diagrammi saranno redatti considerando solo gli aspetti centrali del sistema, ignorando alcuni dettagli tecnici dipendenti ad esempio dal linguaggio `C++`

Per brevità, tali diagrammi documenteranno solo ed esclusivamente metodi pubblici, tralasciando i meotdi a visibilità ristretta e gli attributi di stato interno delle varie classi. Per facilitare la fruizione del documento da parte di coloro non padroneggiano `C++`, le firme dei metodi saranno semplificate (utilizzando dei nomi più User-friendly per i tipi).

## 2.1 Model/Typesystem