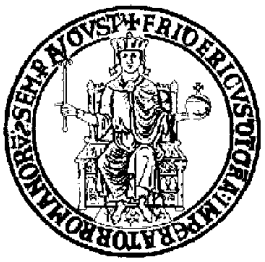
Università degli Studi di Napoli
Federico II



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<Int>(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<Int>(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).

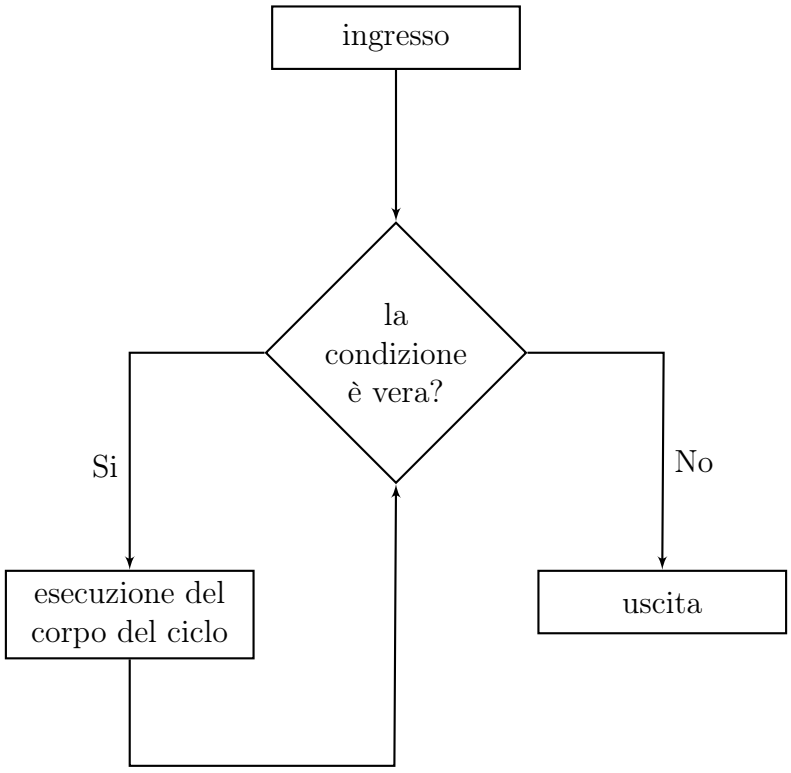


Figura 1: Diagramma di flusso del ciclo while

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).

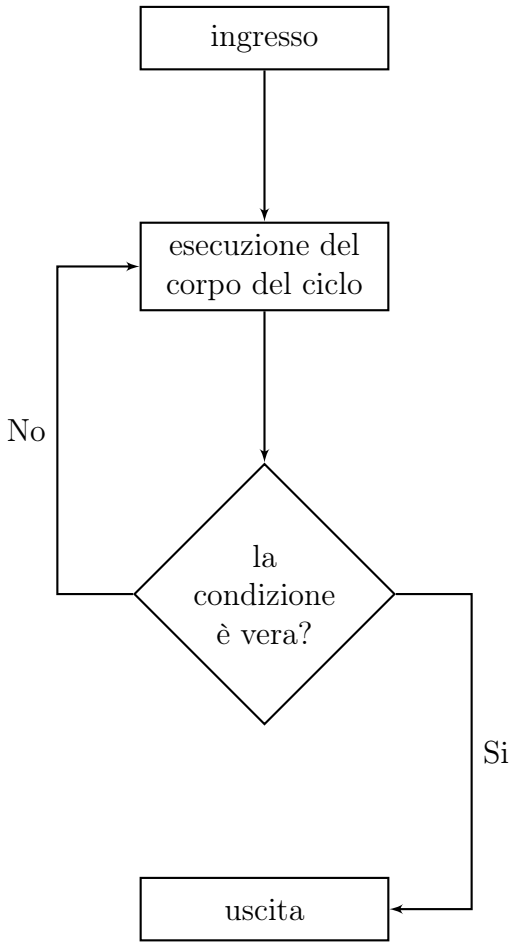


Figura 2: Diagramma del ciclo until

1.3.4 Break e continue

La keyword `break` consente di provocare l'interruzione anticipata do un ciclo. Essa è pensata per essere utilizzata assieme ad un branch condizionale che monitori una qualche condizione eccezionale che richiede l'interruzione immediata del ciclo.
Ad esempio, si analizzi il seguente frammento di codice che illustra un ciclo until:

```
while (true) {
    var x = math.random<Int>(-5,5);
    if (x % 3 == 0) {
        break;
    }
    console.println(x);
}
```

Tale ciclo presenta una condizione da controllare prima della stampa in console, ovvero la non divisibilità per 3 del valore contenuto nella variabile x. Qualora tale condizione si verificasse si uscirebbe immediatamente dal ciclo, altrimenti si procederebbe con la stampa in console del valore di x.

La keyword `continue`, similmente alla keyword break, consente di alterare il flusso di esecuzione di un ciclo. Anzichè provocarne l'interruzione anticipata, essa consente di saltare l'esecuzione del codice rimanente all'interno del corpo e passare direttamente alla successiva iterazione. Ad esempio, si consideri il seguente frammento di codice:

```
var x : Int = 0;
while (x < 10) {
    if (x % 3 == 0) {
        continue;
    }
    console.println(x);
    x = x + 1;
}
```

L'esecuzione di questo ciclo avrò come effetto la stampa in console dei numeri da 0 a 9 che non sono divisibili per 3, in quanto ad ogni iterazione dove x avrà valore divisibile per 3, la stampa in console sarà saltata e si proseguirà all'iterazione seguente.

1.4 Tipi primitivi

Il sistema dei tipi, spesso più comunemente chiamato Typesystem, è un insieme di regole che definiscono il comportamento e le operazioni consentite su tipi di dati. Questo sistema è fondamentale per garantire la correttezza e la sicurezza del codice, limitando gli errori di tipo durante la compilazione o l'esecuzione del programma. In Basalt, tale sistema prevede un insieme di tipi detti tipi primitivi, i quali esistono nativamente nel linguaggio, e permette all'utente di definire tipi personalizzati.

1.4.1 Tipi primitivi semplici

Con "tipi primitivi semplici", in Basalt, si intendono i seguenti tipi di dato:

IDENTIFICATIVO	***DESCRIZIONE***
`Int`	tipo di dato preposto alla rappresentazione dei numeri interi, rappresentato a 64 bit
`Float`	tipo di dato preposto alla rappresentazione dei numeri decimali frazionari, internamente analogo ad un double in C/C++
`Bool`	tipo di dato preposto alla rappresentazione di valori logici (booleani) di vero/falso
`Char`	tipo di dato preposto alla rappresentazione di un singolo carattere ascii 8 bit

Tabella 2: Tipi primitivi

In Basalt, variabili il cui tipo è un tipo primitivo semplice, vengono allocate su stack. Tutte le volte che si lavora con una variabile così dichiarata, si deve dunque assumere che essa si trovi o sullo stack della funzione corrente (compresi gli argomenti delle funzioni).

1.4.2 Array

In Basalt, gli array sono dei blocchi di memoria contigua, capaci di contenere un numero noto a tempo di compilazione di oggetti dello stesso tipo.

Dato un tipo Type ed una lunghezza N allora il tipo [N]Type denoterà il tipo di un array contente esattamente N oggetti di tipo Type. Basalt conserva la lunghezza come parte del tipo, ciò implica che è possibile definire una funzione che prenda come parametro di input un array di cui sia specificata la lunghezza, a differenza del C dove invece si è obbligati a passare la lunghezza tramite l'utilizzo di un parametro ausiliario.

Basalt supporta array-literals sottoforma di tipo esplicito dell'array, seguito da una lista di valori separati da virgole e racchiusi tra parentesi graffe. Tale sintassi può essere usata per inizializzare un array in sede di dichiarazione come illustrato di seguito.

```
var array : [10]Int = [10]Int{0,1,2,3,4,5,6,7,8,9}
```

Così come in quasi tutti i linguaggi imperativi ad oggi usati, dato un array, si può accedere in lettura (e in scrittura qualora non sia costante) al suo ennesimo elemento usando la canonica sintassi storicamente introdotta dal C, che prevede di postporre all'espressione costituente l'array e racchiusa tra parentesi quadre, una espressione il cui valore sia intero e che corrisponda alla posizione dell'elemento all'interno dell'array, assumendo un'indicizzazione che parte da zero.

In generale, un array occupa in memoria un numero di byte pari al prodotto della dimensione in byte di un singolo oggetto in esso conservato, moltiplicato per la lunghezza, ed è dunque privo di qualunque overhead dato che la dimensione è nota a tempo di compilazione e pertanto non viene conservata in memoria.

Un assignment tra array è possibile solo se hanno la stessa dimensione e se i tipi degli oggetti in essi conservati sono tali da consentire un ipotetico assegnamento cella a cella. Qualora tali requisiti siano soddisfatti allora l'assignment performerà una copia di tutti gli elementi dell'array sorgente nell'array destinazione.

1.4.3 Puntatori scalari

In Basalt, i puntatori scalari, più semplicemente detti puntatori, sono dei riferimenti ad un oggetto allocato in memoria, avente un certo tipo noto a tempo di compilazione.

Dato un qualunque tipo `T`, allora con `#T`, indichiamo il tipo dei puntatori a oggetti di tipo `T`. In Go, C e C++ il simbolo preposto a questo scopo è l'asterisco ("∗"), mentre in Jai il simbolo preposto a questo scopo è il carot ("^"). Il motivo per cui Basalt si discosta dagli altri linguaggi per quanto riguarda il simbolo usato per indicare un puntatore è che Basalt vuole cercare di non usare lo stesso simbolo in contesti troppo diversi fra loro. In particolare, dato che l'asterisco e il carot sono simboli già in uso in qualità di operatori binari, è sembrato più saggio scegliere un altro simbolo da dedicare allo scopo di indicare i puntatori.

In maniera conforme a quanto visto in C, C++, Go e molti altri linguaggi, dato un qualunque oggetto di tipo Type, l'operatore unario prefisso `&` consente di estrarre l'indirizzo di memoria di tale valore. Tale indirizzo avrà tipo `#T` e sarà per tanto assegnabile ad un puntatore a Type come mostrato nel seguente esempio.

```
var number : Int = 6;
var ptr : #Int = &number;
```

Ad un puntatore, è possibile assegnare un valore fittizio detto null per rappresentare il fatto che in quel momento il puntatore non sta puntando a un'area di memoria valida.

I puntatori possono riferirsi sia ad aree di memoria su stack sia su heap, ma per allocare memoria su heap sarà necessario chiamare manualmente funzioni di allocazione. Una volta allocata memoria, essa dovrà essere deallocata manualmente in quanto Basalt non possiede un garbage collector a differenza di Go e Java, e invece consente all'utente di gestire la memoria manualmente così come C, C++, Zig, Odin e Jai.

Nel package memory è possibile trovare una funzione malloc e una funzione free, preposte all'allocazione e alla deallocazione di memoria dinamica su heap, di seguito è riportato un esempio d'uso. Si tenga a mente che la sintassi con le parentesi angolari sarà analizzata con maggior dettaglio in seguito nella sezione dedicata ai generics.

```
var ptr : #Int = memory::malloc<Int>(6);
memory::free<Int>(ptr);\vspace{0.5cm}
```

1.4.4 Puntatori vettoriali

Contrapponendosi ai puntatori scalari, vi sono poi i puntatori vettoriali. Un puntatore vettoriale è un puntatore ad una sequenza di oggetti contigui in memoria il cui tipo è noto a tempo di compilazione, ma la cui lunghezza è nota a tempo di esecuzione. Per semplicità è possibile chiamarli "slice" così come si fa in molti altri linguaggi.

I puntatori vettoriali sono internamente implementati come una coppia di un puntatore ed una dimensione. Dato un tipo T allora il tipo $T ne denoterà il puntatore vettoriale.

Un puntatore vettoriale in una macchina a 64bit occupa internamente 16 byte, di cui 8 sono dedicati a conservare un indirizzo di memoria ed altri 8 sono dedicati a conservare la lunghezza, ovvero il numero di celle contigue allocate a partire da tale indirizzo.

A differenza dei puntatori scalari, un puntatore vettoriale non può essere null, però può avere dimensione zero, che è infatti il comportamento standard per un puntatore vettoriale non ancora inizializzato. Questo consente di poter scrivere codice che lavora con puntatori vettoriali senza doversi assicurare ogni volta che il puntatore sia non nullo, ma semplicemente controllando di accedere sempre ad esso con indici strettamente minori della sua dimensione come è del resto naturale fare anche per gli array.

La sintassi per accedere all'i-esimo elemento di un puntatore vettoriale è del tutto uguale a quanto già visto per gli array, ovvero si pone alla destra del puntatore vettoriale da cui si desidera leggere, un espressione di tipo intero il cui valore numerico sarà interpretato come indice racchiuso fra parentesi quadre.

Il seguente frammento di codice illustra come si può istanziare un blocco di memoria dinamica su heap e come lo si può gestire mediante un puntatore vettoriale a tale blocco. In particolare il seguente codice stampa il contenuto di ogni cella del blocco.

```
var i : Int = 0;
var slice : $Int = malloc<$Int>([5]Int{0, 1, 2, 3, 4});

while (i < slice.size){

    println(slice[i]);
    i = i + 1;
}
memory::free<$Int>(slice);
```

É possibile assegnare ad una variabile di tipo "puntatore vettoriale a T", un'espressione di tipo "array di oggetti di tipo T" di qualsiasi dimensione. Ciò consente l'utilizzo del puntatore vettoriale come supertipo di tutti gli array. Tale assegnazione comporta un effetto simile a quello osservato quando si assegna l'indirizzo di un oggetto già istanziato a un puntatore utilizzando l'operatore di indirizzo &. In tal modo, entrambi i riferimenti puntano alla stessa area di memoria.

```
var array : [10]Int = [10]Int{0,1,2,3,4,5,6,7,8,9};
var slice : $Int = array;
```

Dato che un puntatore vettoriale, così come un puntatore scalare, non conserva informazioni sufficenti a determinare se l'oggetto puntato si trovi su stack o su heap, e dato che Basalt si prefige come obbiettivo quello di non effettuare allocazioni nascoste e invece di essere sempre trasparente riguardo alla gestione della memoria, ne consegue che non è posibile inserire nuovi elementi in un puntatore vettoriale o ridimensionarlo in qualsiasi altro modo.

Un'ipotetica implementazione di un array dinamico propriamente detto con possibilità di inserire e rimuovere elementi da esso potrebbe essere quella mostrata nel seguente frammento di codice. Si tenga a mente che tale frammento usa struct e generics, entrambi argomenti che saranno trattati in dettaglio nelle loro sezioni apposite.

```
package slicedemo;

struct Slice<T> {
    storage : $T;
    size : Int;
}

func append<T>(slice : Slice<T>, value : T){
    if (slice.size + 1 > slice.storage.length){
        var old : $T = slice.storage;
        var new_length = 2 * slice.storage.length;
        slice.storage = memory::malloc<$T>(new_length);
        memory::copy<T>(old, slice.storage);
        memory::free<$T>(old);
    }
    slice.storage[slice.size] = value;
    slice.size += 1;
}
```

1.4.5 Stringhe

La gestione delle stringhe nei linguaggi di basso livello è da sempre una sfida, in C, C++, Zig e Odin le stringhe non sono altro che puntatori ad aree di memoria contigue dove sono conservati dei caratteri. Tale è anche l'approccio di Basalt, dove le stringhe, indicate con String, sono implementate come puntatori vettoriali a carattere.

Per facilitare l'ineroperabilità con C, esiste anche il tipo RawString che è invece implementato come un puntatore scalare a carattere, e che punta al primo carattere della sequenza che compone la stringa. In C infatti, una stringa altro non è che un puntatore al primo carattere che ne fa parte. Non avendo una dimensione, le stringhe in C devono essere marcate al termine da un carattere speciale '\0' che ne segnala la terminazione. Al fine di poter convertire agevolmente una String in una RawString, la quale può essere usata per interfacciarsi con C, allora in Basalt è comunque presente il carattere speciale '\0' al termine di ogni sequenza di caratteri conservata in ogni oggetto di tipo String anche se superfluo. Si analizzi dunque il seguente codice.

```
var str : String = "hello world!";
var cstr : RawString = str;
```

Nel frammento di codice appena mostrato, si assegna il valore di una variabile di tipo String ad una variabile di tipo RawString. É possibile descrivere graficamente lo stato della memoria al termine dell'esecuzione di questo frammento di codice con un Memory-Layout-Diagram nel seguente modo:

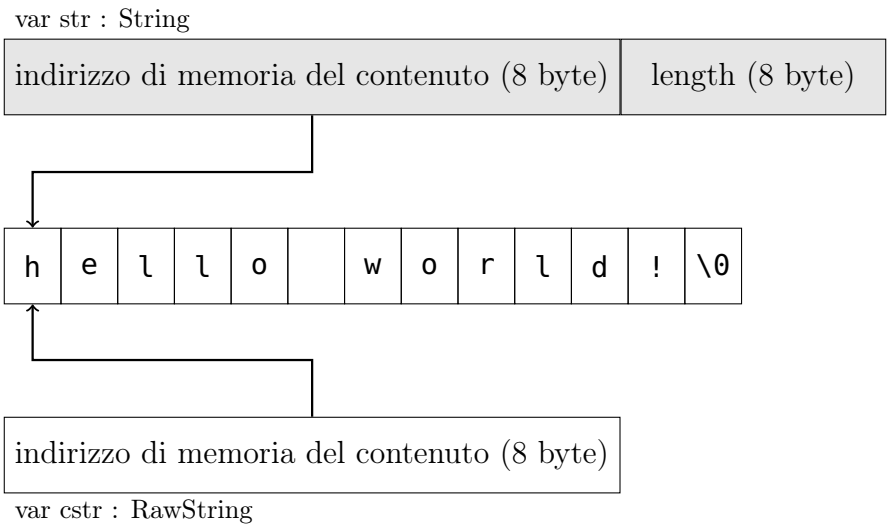


Figura 3: Memory layout dei tipi String e RawString

Qualunque string-literal, quindi anche "Hello, World!" nell'esempio di prima, viene implicitamente spostata nello scope globale così che dall'interno di una funzione si possa restituire una string-literal senza temere che alla fine della chiamata lo stack della funzione venga ripulito e che la stringa appena restituita venga sovrascritta o invalidata.

Questo meccanismo di gestione delle string-literals viene chiamato string-pooling, e l'area di memoria nello scope globale dedicata a contenere tutte le string-literal dell'intero programma viene detta string-pool. Questo meccansimo consente poi di non dover replicare le string-literal, infatti qualora una stessa string-literal apparisse più volte nel programma in scope diversi, sarebbe comunque utilizzato l'indirizzo della stessa unica string-literal nella pool per inizializzare variabili o per effetturare accessi in lettura.

Così come in Go e in Java, le stringhe sono immutabili, questo è un prerequisito essenziale al funzionamento dello string-pooling, dato che stringhe in scope diversi si riferiscono in realtà alle stesse sequenze di caratteri nella pool.

Per modificare una stringa, occorrerà dunque allocare una nuova area di memoria (su stack utilizzando un array di caratteri da castare successivamente a puntatore vettoriale o su heap utilizzando una funzione di allocazione e gestendo un puntatore vettoriale direttamente) per ospitare i caratteri della stringa, si procederà alla modifica e si assegnerà il puntatore vettoriale dell'area di memoria contenete la nuova sequenza di caratteri modificata alla stringa che si desiderava modificare. In Go accade qualcosa di sostanzialmente analogo. Segue un conciso esempio concreto di quanto appena detto.

```
var str : String = "some text";
var tmp : $Char = memory::malloc<Char>(str);

var i : Int = 0;
while (i < tmp.length){
    tmp[i] = uppercase_character(tmp[i]);
    i += 1;
}

str = tmp;
console::println(str);
memory::free<String>(str);
```

Si noti come in questo caso, dato che la stringa ora conserva un riferimento ad un'area di memoria dinamica, allocata su heap manualmente, si rende dunque necessario effettuare una deallocazione manuale al termine dell'utilizzo con la funzione free.

1.5 Struct

Le struct, abbreviazione di "structures" in inglese, rappresentano un fondamentale costrutto di molti linguaggi di programmazione, incluso Basalt. Una struct è difatti un tipo definito dall'utente, preposto alla rapprsentazione di entità complesse, concettualmente rappresentabili come un aggregato di dati distinti.

In altri linguaggi, costrutti analoghi sono chiamati "records" o "product-types".

In Basalt, la definizione di una struct avviene utilizzando la parola chiave "struct" seguita dal nome della struct, che deve iniziare per maiuscola come ogni altro tipo nel linguaggio, e da una serie di campi all'interno di parentesi graffe.

Ogni campo deve essere nella forma <nome> : <tipo>, come mostrato di seguito:

```
struct Person {
    name : String;
    surname : String;
    occupation : String;
}
```

Una volta definita una struct, è possibile usare il suo nome nei contesi dove il linguaggio Basalt richiede un tipo, come ad esempio nella dichiarazione di una variabile, nei parametri delle funzioni o come tipo di un campo di un'altra struct.
Su ogni variabile il cui tipo è una struct, è possibile applicare l'operatore binario "." così come nella maggior parte dei linguaggi di programmazione, tale operatore consente di accedere ai campi uno specifica istanza di una struct.
Assumendo dunque di avere accesso alla definizione di Person dal precedente frammento di codice, sarà dunque lecito dichiarare variabili di tipo Person e accedere in lettura e scrittura ai loro campi utilizzando l'operatore "." avendo come operatore sinistro un oggetto di tipo Person e come operatore destro il nome di uno specifico campo.

```
var john : Person;

john.name = "John";
john.surname = "Doe";
john.occupation = "Programmer";
```

1.5.1 Puntatori a struct

In C e in C++, per accedere ai campi di un oggetto dato un puntatore a tale oggetto, occorre o deferenziare il puntatore, per poi utilizzare l'operatore "." sull'oggetto così ottenuto, oppure è stato introdotto un operatore apposito "->" funzionalmente analogo.

In Basalt, così come in Go, è possibile usare l'operatore "." direttamente sul puntatore per accedere ai campi dell'oggetto puntato. Tale sintassi non porta ambiguità dato che non vi sono altri significati per l'operatore "." applicato ad un puntatore.

Consideriamo infatti il seguente frammento di codice, dove vengono definite diverse variabili, e alcune delle quali sono puntatori. In questo esempio, sarà fatto riferimento alla definizione per la struct Person data nella pagina precedente.

```
var person : Person;
var person_ptr : #Person = &person;
var person_ptr_ptr : ##Person = &person_ptr;
```

Allora si potrà accedere al campo "name" della variabile "person" postponendo ".name" a una qualunque di queste tre variabili.

Go è stato il primo linguaggio ad introdurre un meccanismo del genere, seppur in una forma più limitata dove è possibile accedere al campo dell'oggetto puntato solo da un puntatore che vi punti direttamente, e non consentendolo invece nei casi dove vi sono puntatori a puntatori, o più in generale, due o più livelli di indirezione.

Tale sovraccarico della semantica dell'operatore ".", consente di ridurre al minimo le modifiche da effettuare ad un blocco di codice funzionante qualora si voglia decidere di cambiare il tipo di una delle variabili che esso utilizza rendendola un puntatore invece che un oggetto locale. Dunque la scelta di estendere l'utilizzo di tale operatore in tal modo è stata fatta per facilitare refactoring del codice.

Ciò è particolarmente vero nei casi in cui una funzione utilizza già un puntatore per accedere in lettura e scrittura ai campi di un oggetto e ci si accorge in un secondo momento che tale funzione ha bisogno di eventualmente riassegnare un nuovo valore all'oggetto stesso. In tal caso, l'unica modifica da apportare alla funzione sarà cambiare il tipo dell'argomento in questione e rendendolo un puntatore a puntatore, mantenendo il resto del codice intatto. Chiunque abbia programmato C abbastanza a lungo questo si potrà facilmente rendere conto che tale scenario è molto comune e pertanto facilitare la risoluzione di un problema del genere è qualcosa di cui il programmatore medio può beneficiare in modo cocnreto e tangibile.

1.5.2 Struct ricorsive

Le variabili, al momento della creazione sia su stack che su heap, devono avere una dimensione in bytes nota a tempo di compilazione. Tale dimensione, per le variabili il cui tipo è una struct, è ottenuta calcolando la somma delle dimensioni dei field, i cui tipi possono potenzialmente essere anch'essi struct.

Date queste premesse, è chiaro che definizioni ricorsive come la definizione seguente, sono errate e portano ad un errore a tempo di compilazione.

```
struct Recursive {
    recursive : Recursive;
}
```

Il compilatore Basalt, non potrebbe calcolare la dimensione in byte di un ipotetico oggetto il cui tipo è Recursive e di conseguenza, causa un errore di compilazione.

Basalt riesce ad identificare questo errore esplorando il grafo orientato che le definizioni di struct implicitamente descrivono ed implementa un controllo di aciclicità su di esso.

Basalt in tale controllo si limita ad esplorare gli archi relativi a tipi semplici e array, e invece non esplora archi relativi a puntatori scalari e vettoriali. Questo perchè un puntatore, vettoriale o scalare, ha sempre dimensione nota. Ne consegue che questa definizione alternativa della struct Recursive è invece corretta e perfettamente valida.

```
struct Recursive {
    recursive_ptr : #Recursive;
    recursive_slice : $Recursive;
}
```

1.5.3 Struct generiche

É possibile parametrizzare una definizione di una struct con dei parametri formali di tipo, in gergo detti "generics". Al momento dell'utilizzo di tale definizione, dei parametri attuali di tipo dovranno essere forniti, e una definizione ad-hoc sarà generata da Basalt tramite match-and-replace dei parametri formali all'interno della definizione, performata eventualmente in modo ricorsivo se necessario.

La sintassi per definire una struct con parametri formali di tipo prevede una lista di identificatori di tipo separati da virgole e racchiusi in parentesi angolari alla destra del nome della struct. Di seguito viene riportata la definizione di una linked list doppiamente puntata e parametrica sul tipo di dato conservato in ogni nodo.

```
package listdemo;

struct List<T> {
    size : Int;
    head : #Node<T>;
    tail : #Node<T>;
}

struct Node<T> {
    item : T;
    next : #Node<T>;
    prev : #Node<T>;
}
```

L'implementazione interna a Basalt delle definizioni generiche, che consiste nel generare nuove definizioni ad-hoc per ogni possibile scelta dei parametri attuali di tipo forniti al momento dell'utilizzo è detta reificazione a tempo di compilazione.

1.6 Union

Le union sono un costrutto che consente al programmatore di definire un tipo di dato la cui rappresentazione interna può variare nell'ambito di un numero finito di opzioni mutuamente esclusive e note a priori.

Le union in Basalt non sono implementate come in C, e sono invece più simili ad i "sum-types" presenti in molti linguaggi funzionali come Haskell, Idris o ML.

In Basalt, la definizione di una union avviene utilizzando la parola chiave "union" seguita dal nome della union, che deve iniziare per maiuscola come ogni altro tipo nel linguaggio, dal simbolo uguale, e da una serie di tipi separati da "|" (pipe).

```
union Number = Int | Float
```

Non è necessario definire una union dandole un nome, è infatti possibile utilizzare union anonime, ovvero union definite su una singola riga direttamente al momento dell'utilizzo.

La sintassi per fare ciò, prevede semplicemente di utilizzare una serie di tipi separati da "|" in tutti i contesti in cui il type-system richiede l'utilizzo di un tipo. In automatico tale entità verrà interpetata come union-anonima.

```
var named_union_example : Number = 3.14;
var inline_union_example : Int | Float = 7;
```

1.6.1 Union generiche

Così come le struct, anche le union possono essere generiche (se non anonime), ovvero possono avere parametri formali di tipo. Anche nel caso delle union la loro implementazione concreta consiste nella reificazione a tempo di compilazione.

Un caso particolarmente indicativo dell'utilità di questo costrutto è ad esempio una ipotetica union Collection, generica con parametro di tipo T, definita a partire da una serie di tipi definiti come struct, i quali implementano varie strutture dati.

```
union Collection<T> = LinkedList<T> | HashTable<T> | Tree<T>
```

1.6.2 Assignment tra union

Mentre per le struct, si applica name-equivalence, ovvero è possibile assegnare un valore ad una variabile il cui tipo è una struct solo se il valore è dello stesso identico tipo, per le union si applica una politica di structural-compatibility, ovvero è possibile assegnare alle variabili il cui tipo è una union ogni valore che quella union può rappresentare.

Se è possibile assegnare a variabili il cui tipo è una union, un valore il cui tipo è un'altra union, allora quest'ultima si dice strutturalmente compatibile con la prima.

É possibile dunque assegnare ad una variabile il cui tipo è una union:

- valori del suo stesso tipo
- valori il cui tipo compare esplicitamente nel suo elenco dei tipi
- valori il cui tipo è assegnabile ad un tipo elencato nel suo elenco dei tipi
- valori il cui tipo è un'altra union, definita a partire da tipi a loro volta assegnabili

1.6.3 Assignment tra puntatori a union

Per le variabili di tipo puntatore a union, sia scalari che vettoriali, le regole di assignment sono più stringenti. Ad una variabile di tipo puntatore a union, è possibile assegnare un puntatore ad un'altra union, solo se tali union sono in relazione di structural-equivalence tra loro.

Due union sono in relazione di structural-equivalence tra loro, se l'una è in relazione di structural-compatibility con l'altra, secondo la nozione di structural compatibility data nel pragrafo immediatamente precedente.

La relazione di structural-compatibility, nonostante la sua formulazione apparentemente debole, garantisce che il memory layout interno sia il medesimo, e pertanto anche strumenti quali i puntatori, che sono per definizione dipendenti dalla conoscenza del memroy layout dei valori puntati possono godere di una flessibilità maggiorata.

1.6.4 Union come parametri attuali di tipo

Due tipi, i quali sono istanze della stessa struct con parametri attuali di tipo diversi, sono considerati lo stesso tipo ai fini della name-equivalence e quindi della validazione degli assignment, qualora i parametri di tipo siano in relazione di structural-equivalence a due a due fra loro.

Ad esempio `List<Int|Float>` è assegnabile a `List<Number>` e viceversa, in quanto la union `Number` è in relazione di structural-equivalence con `Int|Float`.

1.6.5 Operatore `is`

Per conoscere il tipo effettivo rappresentato in un certo momento dell'esecuzione del programma da un oggetto il cui tipo è una union, si può utilizzare l'operatore is, il quale si comporta in modo analogo ad instanceof in java o all'omonimo operatore is in C#, ovvero restituisce true se e solo se il tipo concreto dell'oggetto fornito come operando sinistro è assegnabile al tipo fornito come operando destro.

```
var num : Int | Float = 6;

if (num is Int) {
    console::println("num is an integer");
}
else {
    console::println("num is a float");
}
```

Così come le struct, anche le union devono avere una dimensione in byte nota a tempo di compilazione, e dunque analogamente a quanto visto per le struct, Basalt reputa non corrette tutte le definizioni di union aventi dipendenze cicliche dirette.

Una union occupa in memoria dimensione pari al massimo tra le dimensioni dei tipi non-union a partire dai quali è stata definita, con un overhead aggiuntivo di otto bytes utilizzati per contenere l'indirizzo in memoria di un oggetto di tipo Type il quale rappresenta il tipo effettivo rappresentato da un certo oggetto di tipo union in un dato momento dell'esecuzione del programma. Il tipo Type è un tipo speciale in Basalt, esso è la spina dorsale su cui è stato costruito il meccanismo di reflection, e ad esso è stata dedicata una sezione apposita di questo documento.

Alla luce di quanto detto, ogni oggetto di tipo PrimitiveType dunque occuperà in memoria sedici bytes, otto dedicati a rappresentare il valore da esso incapsulato, che può essere di tipo Int, Float, Char, Bool o Byte, ed otto per rappresentare l'indirizzo di un oggetto tipo Type, che descrive il tipo di tale valore incapsulato.

Si noti come in altri linguaggi, come Java, Kotlin e C#, ogni oggetto possiede un "object-header", ovvero un'area del proprio memory layout atta a contenere varie informazioni, tra le quali il tipo effettivo dell'oggetto. La differenza principale tra tali linguaggi e Basalt è che quest'ultimo limita l'overhead alle sole union, che infatti sono il solo costrutto che consentono il disallineamento tra tipo dichiarato e tipo effettivo.

1.6.6 Operatore `as`

Per poter accedere al valore internamente contenuto da una variabile il cui tipo è una union è possibile usare l'operatore as, operatore binario il cui operando sinistro è una espressione il cui tipo è una union, mentre l'operando destro è un tipo che si desidera estrarre dalla union.

L'operatore as, è utilizzabile solo su un tipo che sarebbe teoricamente assegnabile all'espressione sulla quale esso viene usato, pena un errore a tempo di compilazione.

Se lo si usa su espressioni che contengono un tipo diverso, esso non fallisce a tempo di esecuzione, ma si limita a fornire valori indefiniti corrispondenti all'interpretazione dei byte del contenuto reale della union come se essi fossero invece del tipo richiesto.

L'uso dell'operatore as è consigliato solo all'interno di dei branch condizionali, o dopo degli assert, la cui condizione assicura che la union contenga effettivamente il tipo che il programmatore si aspetti a tempo di esecuzione.

L'operatore as fornsice un vero e proprio riferimento utilizzabile non solo in lettura ma anche in scrittura, è analogo al reinterpret-cast di C++ ma, se usato in condizioni in cui l'operatore is con gli stessi operandi avesse valore true, allora il suo buon funzionamento è sempre garantito.

1.6.7 Union ricorsive

Una union, così come una struct, deve avere una dimensione in byte nota a tempo di compilazione, e tale dimensione è funzione delle dimensioni dei tipi a partire dai quali essa è definita. Analogamente a quanto visto per le struct dunque, la seguente definizione non è valida in quanto Basalt non è in grado di calcolare la dimensione di una ipotetica variabile di tipo Recursive.

```
union Recursive = Int | Recursive
```

Per gli stessi motivi per cui ciò era valido per le struct, è però possibile definire union ricorsive con la ricorsione indiretta, ovvero usando puntatori (vettoriali e scalari).

```
union Recursive = #Recursive | $Recursive
```

1.6.8 Memory-layout di una union

Come già detto nel paragrafo precedente, è stato detto che la dimensione in Byte occupata da una union, è calcolata in funzione della dimensione del tipo con la dimensione più grande tra quelli a partire dalla quale essa è stata definita.

Una union è internamente rappresentata come due blocchi di byte adiacenti in memoria, il primo, di 8 byte, è detto header, ed è usato per contenere metadati necessari al corretto funzionamento dell'operatore is, mentre il secondo è detto payload, e contiene la rappresentazione in byte del valore rappresentato a tempo di esecuzione dalla union.

Figura 4: Memory layout dei tipi String e RawString

Definiamo dimensione netta di un tipo la dimensione del suo payload, se esso è una union, o la sua dimensione complessiva in byte se esso è un tipo di altra natura.

La dimensione in byte del payload di una union, ovvero la sua dimensione netta, è pari alla dimensione netta del tipo con dimensione netta maggiore tra quelli a partire dai quali la union è stata definita.

Per union definite a partire da altre union dunque, gli overhead dati dagli header non sono cumulativi. Gli 8 byte dedicati all'header sono usati per conservare l'indirizzo in memoria a cui sono conservate le type informations relative al tipo di volta in volta contenuto all'interno della union. In sede di assignment, che è l'unica occasione in cui il tipo contenuto possa cambiare, tale puntatore viene eventualmente aggiornato.

L'assignment ad una union quindi è in realtà una coppia di due operazioni, la prima è la scrittura dei byte all'interno del payload (nel caso in cui il tipo del valore assegnato sia una union, saranno copiati solo i byte del payload), la seconda è la scrittura dei byte relativi all'header con l'indirizzo, staticamente noto, delle type-informations del tipo che si è andati ad assegnare (nel caso in cui tipo del valore assegnato sia una union, saranno copiati i byte del suo header all'interno dell'header della union destinazione).

Qualcosa di funzionalmente analogo a quanto descritto fin ora, sono le std::variant introdotte nella libreria standard C++ a partire dallo standard C++17. Esse non sono parte del core language, e sono invece definite usando la metaprogrammazione C++.

2 Implementazione

2.1 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).

2.1.1 Model/Typesystem

Le classi più importanti del package `model` e del package `typesystem` sono rappresentate nel seguente diagramma UML-Class-Diagram.

Esse sono state progettate per rappresentare le strutture dati fondamentali del compilatore Basalt, quali le varie tabelle atte a conservare in memoria le definizioni di tipi e funzioni, le classi di utilità che si occupano di risolvere gli overload e di generarli tramite CFA.

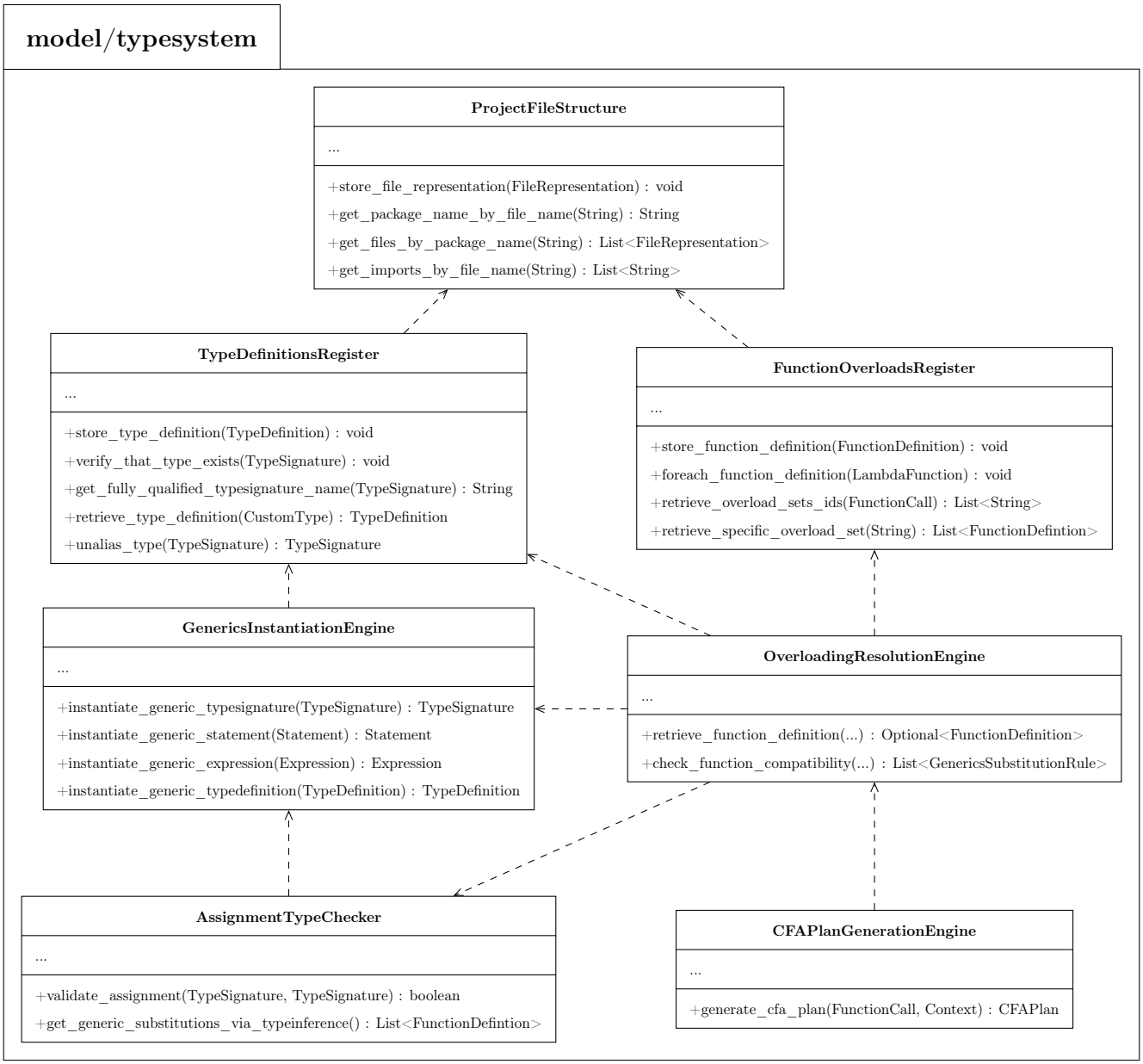


Figura 5: UML-Class-Diagram Model/Typesystem

Per quanto riguarda in particolare il sistema di generazione degli overload tramite CFA, esso modella gli overload auto-generati grazie alla classe `CommonFeatureAdoptionPlanDescriptor`. Essa è implementata tramite un variant, un'approccio tipico in C++ il quale non è facilmente rappresentabile in UML. Per rappresentare tale classe quindi, si è scelto di diagrammare una gerarchia di classi funzionalmente analoga a quanto implementato. Si tenga a mente dunque che il seguente diagramma è solo a scopo illustrativo, ma non riflette fedelmente la struttura interna.

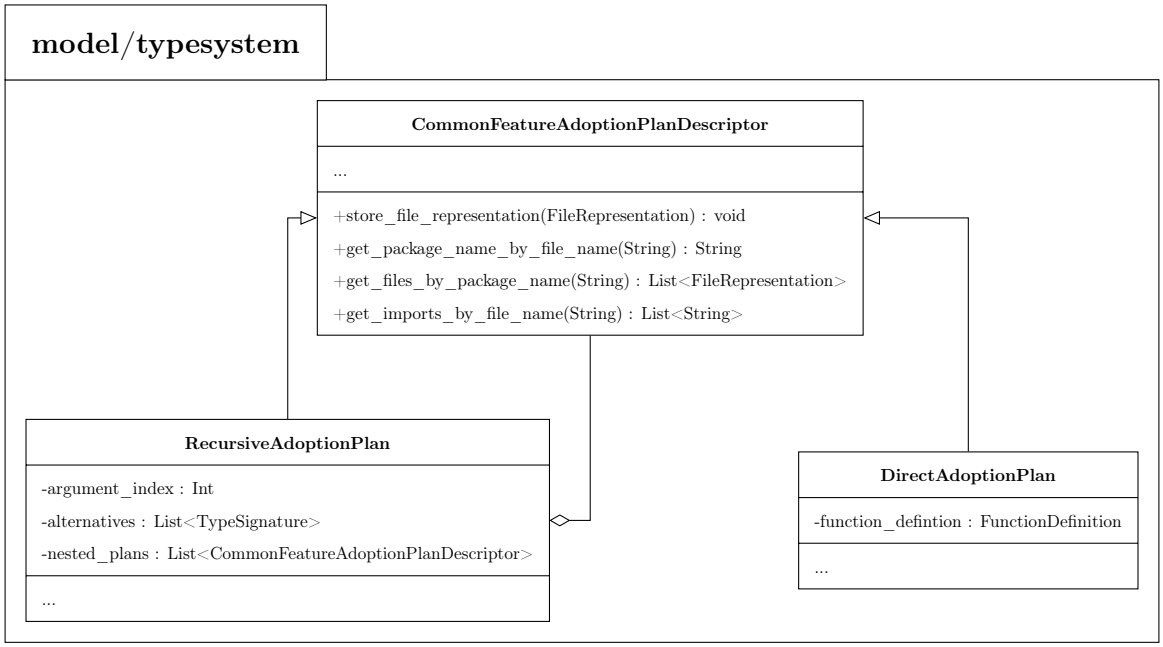


Figura 6: UML-Class-Diagram CommonFeatureAdoptionPlanDescriptor

Tale struttura è atta a rappresentare un albero in memoria, dove ogni nodo foglia è una definizione di funzione, e ogni nodo non-foglia è un dispatch per un certo argomento di una funzione basato sul suo tipo. Per costruzione tale albero è N-ario, con N corrispondente al numero di casi coperti dalla union (anonima o non) che rappresenta il tipo dell'argomento in questione.