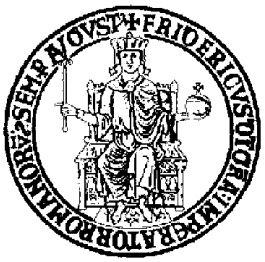
Università degli Studi di Napoli
Federico II



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 un'alternativa moderna a linguaggi di programmazione consolidati come C e C++. Nonostante questi ultimi siano ancora ben lontani dall'essere considerati obsoleti, è innegabile che comincino a mostrare i segni del tempo se paragonati con linguaggi moderni quali Go, Rust, Zig, Odin o Carbon.

L'obiettivo di Basalt è quello di essere semplice e minimale, facile da imparare, rimanendo al tempo stesso un linguaggio di basso livello e pertanto con gestione manuale della memoria. Basalt pone l'ergonomia al centro di ogni scelta di design, cercando di ridurre al minimo il tempo speso dal programmatore a correggere errori banali o scrivere codice ripetitivo.

Basalt, così come i sopra citati Go, Rust, Zig, Odin o Carbon, i quali saranno usati come termini di paragone per tutto il resto del capitolo, non adotta il paradigma ad oggetti. La nuova tendenza nei linguaggi di programmazione sembra essere di abbandonare il paradigma ad oggetti puro, offrendone una versione fortemente rivisitata o addirittura eliminandolo del tutto. Nel caso di Basalt, il ruolo operativamente ricoperto dalle interfacce è stato preso dalle union (sum-types), le quali possono essere usate per offrire funzionalità simil-polimorfiche mediante sostanziali integrazioni con le altre features del linguaggio.

Basalt, a differenza del C, è provvisto di un sistema dei tipi più avanzato, che permette di passare array statici o dinamici alle funzioni senza perdere informazioni riguardanti la dimensione di questi ultimi.

Basalt offre oltretutto pieno supporto alla programmazione generica, implementata mediante reificazione a tempo di compilazione (così come C++), e non mediante erasure, come ad esempio Java o Kotlin. Il supporto alla programmazione generica è stata una tra le prime features ad essere state implementate ed ha guidato molte delle scelte di design del linguaggio.

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 considerare 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	✓	✓	✓	✓	
indipendente da runtime-enviroment di qualsiasi tipo	✓	✓	✓		
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:	10/13	6/13	8/13	6/13	7/13

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'interno di un file appartenente ad un dato pacchetto, sia visibile il contenuto di tutti gli altri file dello stesso pacchetto, assieme al contenuto dei package importati.

Ogni file sorgente contenente codice Basalt deve possedere un'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

Le variabili 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.

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, seguita dal nome della variabile, dai due punti e dal tipo di tale variabile.

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

1.2.2 Scoping di una variabile

Una variabile in Basalt esiste nello scope della funzione, del ciclo o più in generale del blocco di codice in cui è stata dichiarata. Ciò significa che una variabile dichiarata all'interno di un blocco di codice non sarà accessibile al di fuori di esso.

1.2.3 Shadowing

Con shadowing, si intende la possibilità di, all'interno di un blocco di codice innestato, dichiarare una variabile con un nome già usato in un blocco esterno, oscurandola. Numerosi linguaggi supportano lo shadowing delle variabili, ma Basalt non è tra questi. Si è ritenuto che tale funzionalità potesse portare a confusione e a codice di difficile comprensione, pertanto tentare di oscurare una variabile già definita in un blocco esterno causerà un errore a tempo di compilazione.

1.2.4 Deallocazione delle variabili

Al termine dell'esecuzione del blocco di codice corrente, l'area di memoria occupata dalle variabili dichiarate in esso verrà automaticamente deallocata. Nel caso in cui tali variabili abbiano per valore un indirizzo di memoria dinamica allocato in precedenza, la deallocazione di tale blocco **non** è automatica, e spetterà dunque al programmatore deallocare tale blocco di memoria manualmente.

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).

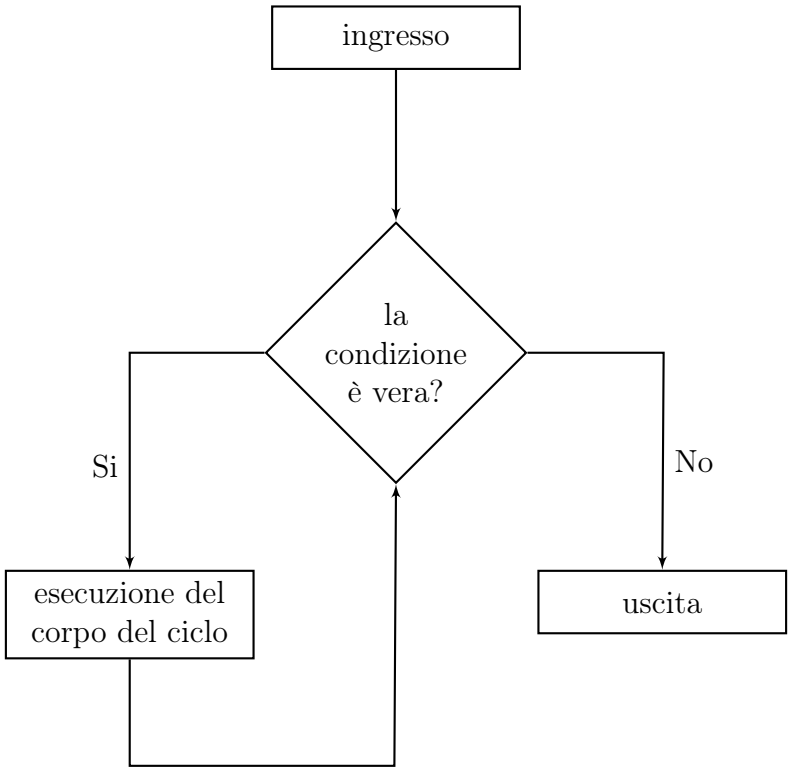


Figura 1: Diagramma di flusso del ciclo while

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 until 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).

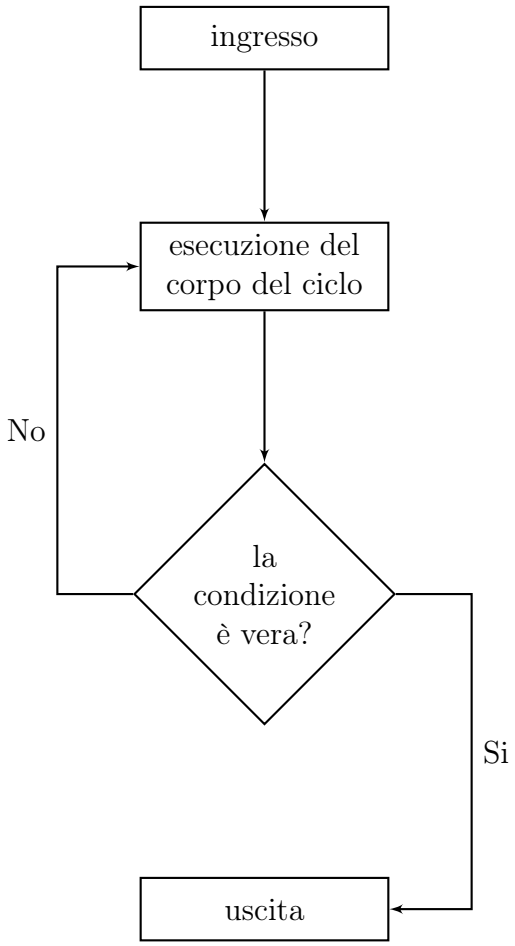


Figura 2: Diagramma del ciclo until

1.3.4 Break e continue

La keyword `break` consente di provocare l'interruzione anticipata da 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 while:

```
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 avrà 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. 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 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 contenente 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 posporre all'espressione costituente l'array e racchiusa tra parentesi quadre, un'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, racchiusa 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 "puntatore scalare ad 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 prefigge come obbiettivo quello di non effettuare allocazioni nascoste e invece di essere sempre trasparente riguardo alla gestione della memoria, ne consegue che non è possibile 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'interoperabilità con C, esiste anche il tipo RawString che è invece implementato come un puntatore scalare a carattere, il quale, 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:

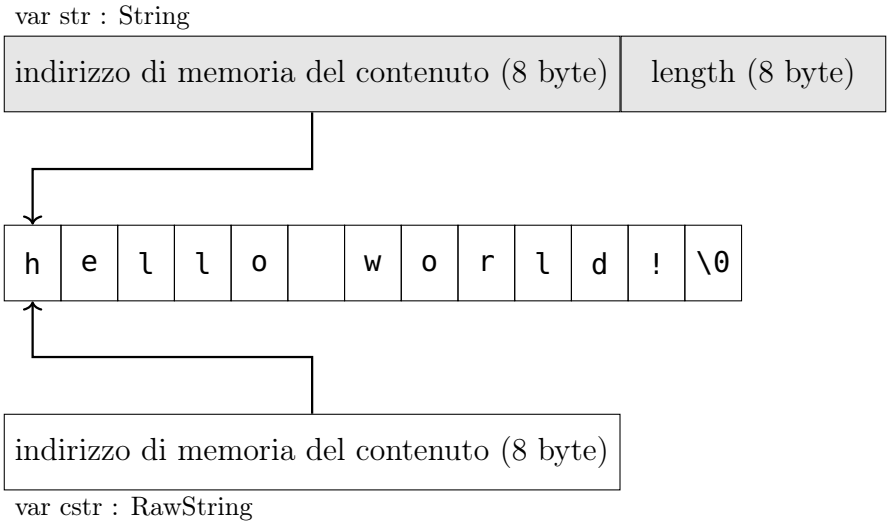


Figura 3: Memory layout dei tipi String e RawString

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 per ospitare i caratteri della stessa (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). Si procederà alla modifica e si assegnerà il puntatore vettoriale dell'area di memoria contenente 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 modellazione 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 contesti 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 di una 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 dereferenziare il puntatore, per poi utilizzare l'operatore "." sull'oggetto così ottenuto, oppure usare l'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" posponendo ".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 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 concreto 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 {

    // Ricorsione diretta -> Errore
    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 {

    // Ricorsione indiretta -> Corretto
    recursive_ptr : #Recursive;
    recursive_slice : $Recursive;
}
```

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 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
```

1.6.2 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++.

1.6.3 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");
}
```

1.6.4 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 un'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.

1.7 Generics

Con generics ci si riferisce a parametri formali di tipo applicabili a definizioni di tipi e funzioni all'interno del linguaggio di programmazione Basalt.

Tali definizioni diventano così parametriche, vengono dunque sottoposte a un type-checking ridotto e vengono utilizzate come dei template per generare definizioni concrete (non-parametriche) al momento del loro utilizzo, istanziandole con i valori concreti di tali parametri di tipo.

Tale approccio all'implementazione dei generics è detto "reificazione" ed è usato da linguaggi come ad esempio `C++`, a tempo di compilazione, e da `C#`, a tempo di esecuzione. Al contrario, linguaggi come Java e Kotlin usano un approccio detto "erasure".

1.7.1 Struct generiche

In Basalt, le struct sono parametrizzabili mediante l'utilizzo dei generics. La sintassi per definire una struct generica 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.

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

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

1.7.2 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>
```

1.7.3 Funzioni generiche

Come detto in precedenza, le funzioni in Basalt possono essere generiche. La definizione di una funzione generica prevede la presenza di una lista non vuota di parametri formali di tipo, separati da virgole e racchiusi tra parentesi angolari, che precede la lista di argomenti della funzione.

È possibile definire una funzione generica che restituisca il massimo tra due valori il cui tipo è specificato al momento della chiamata.

```
func max<T>(first : T, second : T) -> T {
    if (first > second) {
        return first;
    }
    else {
        return second;
    }
}
```

Tale definizione sarà istanziata all'occorrenza e sarà possibile istanziare tale funzione solo per tipi confrontabili con l'operatore '>'. Istanziare tale funzione con tipi non confrontabili genererà un errore a tempo di compilazione.

In sede di chiamata a funzione, è possibile specificare dei parametri attuali di tipo per la funzione stessa dopo il nome e prima dell'elenco degli argomenti, elencandoli separati da virgole e racchiusi tra parentesi angolari.

Ad esempio per la funzione add è possibile usare sia `Int`, che `Float`.

```
var x : Int = max<Int>(3, 5);
var y : Float = max<Float>(3.14, 5.17);
```

1.7.4 Algoritmo di type-inferece

Per funzioni generiche è possibile non specificare espressamente dei parametri attuali di tipo e lasciare che sia Basalt a dedurli dal contesto. L'operazione di deduzione dei parametri attuali di tipo a partire dal contesto è detta type-inference.

Supponiamo ad esempio di voler chiamare la funzione max, ma senza specificare un parametro attuale di tipo. Per usare la type-inference, basterà omettere interamente i parametri attuali di tipo ed utilizzare max come se fosse una funzione non generica.

```
var x : Int = max(3, 5);
var y : Float = max(3.14, 5.17);
```

Consideriamo poi le due seguenti chiamate, dove in entrambi i casi, i tipi degli argomenti forniti in chiamata sono distinti. Dato che nella definizione della funzione `max` compare un unico parametro formale di tipo `T` che risulta essere il tipo di entrambi gli argomenti, l'algoritmo di typeinference si troverà a risolvere due vincoli per un solo tipo.

```
var int_value : Int = 3;
var float_value : Float = 3.14;
var number_value : Number = 5;

var z1 : Number = max(int_value, number_value);
var z2 : Number = max(int_value, float_value);
```

La risoluzione dei vincoli di tipo è affrontata secondo il seguente algoritmo (in flow-chart):

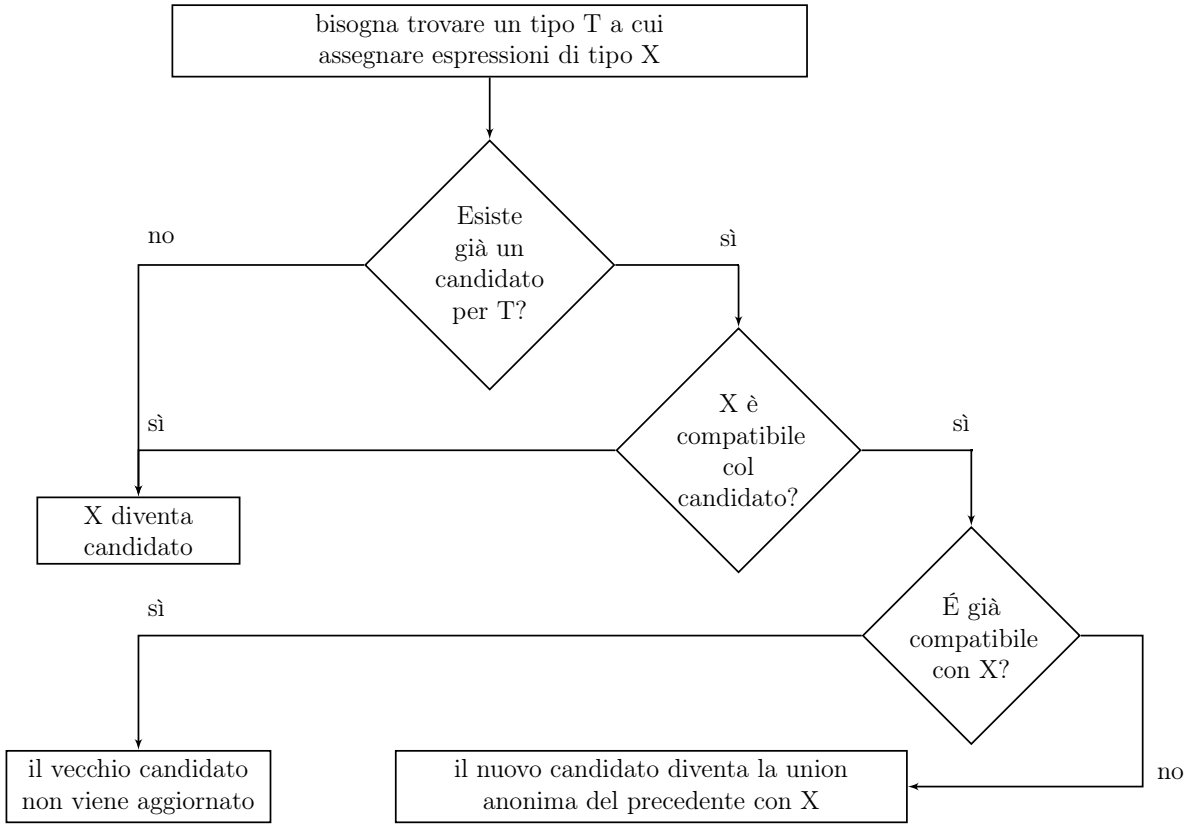


Figura 5: Flow chart per l'algoritmo di type inference

Con tale algoritmo, la chiamata alla funzione `max` con argomenti di tipo `Int` e `Number` porterà alla deduzione del tipo `Number`, mentre la chiamata con argomenti di tipo `Int` e `Float` porterà alla deduzione del tipo `Int | Float` (comportamento atteso).

1.8 Funzioni

Una funzione è un blocco di codice riutilizzabile molteplici volte la cui esecuzione può venire influenzata dal valore di eventuali parametri, qualora presenti, detti argomenti. Ogni argomento ha un tipo ed un nome, con il quale è possibile riferirsi ad esso.

Una funzione può "restituire" un valore al chiamante, ed il tipo di tale valore di ritorno è noto a tempo di compilazione qualora presente. È anche possibile che una funzione non restituisca nulla al chiamante, in tal caso si parla di procedura o di routine.

La definizione di una funzione in Basalt avviene utilizzando la parola chiave func, seguita dal nome della funzione, da un eventuale elenco non vuoto di parametri formali di tipo separati da virgole e racchiusi tra parentesi angolari, da un elenco eventualmente anche vuoto di argomenti, separati da virgole e racchiusi in parentesi tonde, da un eventuale tipo di ritorno preceduto dal simbolo "->" e infine da un blocco di codice delimitato da parentesi graffe chiamato corpo della funzione.

Di seguito è riportato un esempio di definizione di una funzione "max" che accetta due argomenti di tipo Int e restituisce un Int corrispondente al valore maggiore tra loro.

```
func max(first : Int, second : Int) -> Int {
    if (first > second) {
        return first;
    }

    else {
        return second;
    }
}
```

È possibile invocare questa funzione passando una qualunque coppia di valori interi, e più in generale è possibile invocare una funzione con una lista di valori i cui tipi siano compatibili per assegnazione ai tipi degli argomenti di tale funzione. La sintassi della chiamata a funzione in Basalt è equivalente a quanto si può vedere in linguaggi come C, C++ e Go, e consta del nome della funzione, seguito da una eventuale lista di parametri attuali di tipo racchiusa in parentesi angolari e separati da virgole e da una lista di valori, da assegnare agli argomenti, racchiusi tra parentesi tonde e separati da virgole.

```
var n : Int = max(5,6);
```

1.8.1 Overloading

Con overloading delle funzioni si intende la possibilità di fornire all'interno di un programma, eventualmente anche all'interno dello stesso package, multiple definizioni di funzioni aventi lo stesso nome, a patto che gli argomenti differiscano per quantità o per il tipo di almeno uno di essi. Due definzioni di funzioni aventi lo stesso nome si dicono una overload dell'altra. L'insieme di tutti gli overload di una certa funzione viene chiamato overload-set.

Analizziamo per esempio questi due overload per la funzione `max`, esse sono validi overload in quanto pur avendo lo stesso numero di argomenti, tali argomenti hanno tipo distinto, e dunque in sede di chiamata il compilatore analizzando i tipi degli argomenti concreti della chiamata sarà in grado, partendo da essi, di selezionare l'overload adatto.

```
func max(first : Int, second : Int) -> Int {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}

func max(first : Float, second : Float) -> Float {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}

func max(first : Number, second : Number) -> Number {
    if (first > second) {

        return first;
    }
    else {
        return second;
    }
}
```

Nel caso poi in cui esistano due overload entrambi validi, sarà scelto l'overload ritenuto più specifico, applicando queste politiche di selezione e filtraggio tra gli overload trovati.

- un overload non generico viene sempre preferito rispetto ad un overload generico, il numero di parametri di tipo non è rilevante
- un overload viene preferito ad un altro se i suoi parametri di tipo compaiono meno volte nei tipi dei suoi argomenti
- un overload viene preferito ad un altro se i suoi argomenti hanno tipi più complicati, ovvero hanno più parametri di tipo, oppure tali parametri di tipo sono a loro volta, ricorsivamente, più complicati
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti compaiono meno volte tipi definiti come union e/o union anonime
- un overload viene preferito ad un altro se il numero totale di casi coperti dalle union che compaiono tra i suoi argomenti è minore
- un overload viene preferito ad un altro se tra i tipi dei suoi argomenti e/o tra i parametri di tipo di questi ultimi vi sono meno conversioni di tipo

É possibile analizzare tutti gli aspetti appena elencati durante una fase apposita di preprocessing. Le funzioni quindi vengono valutate per la loro specificità prima che la ricerca dell'overload più appropriato abbia inizio. Tale ricerca, nota come overload-resolution è sostanzialmente un'operazione di scarto degli overload non compatibili con i tipi della chiamata, procedendo in ordine di specificità, il cui ordine è stato già stabilito a priori.

Qualora, alla fine della fase di scarto degli overload incompatibili, siano state trovate due definizioni ugualmente specifiche allora la chiamata viene definita ambigua e ciò porta ad un errore a tempo di compilazione.

In particolare, applicando la regola numero 3, si è in grado di affermare che tra i seguenti due overload di seguito riportati, vi è uno più specifico dell'altro ed esso è, come è lecito aspettarsi, l'overload che accetta un argomento di tipo `List<List<T>>`

IDENTIFICATIVO	***SPECIFICITÀ***
`func f<T>(x : List<List<T>>)`	#1
`func f<T>(x : List<T>)`	#2
`func f<T>(x : T)`	#3

Tabella 3: Comparazione specificità di overload per la funzione `f`

Di seguito è stata riportata una tabella dove sono state riportate alcuni overload della funzione `max`, raggruppate per specificità ed elencate dalla più alla meno specifica.

IDENTIFICATIVO	***SPECIFICITÀ***
`func max(first : Int, second : Int) -> Int`	#1
`func max(first : Float, second : Float) -> Float`	#1
`func max(first : Number, second : Int) -> Number`	#2
`func max(first : Int, second : Number) -> Number`	#2
`func max(first : Number, second : Float) -> Number`	#2
`func max(first : Float, second : Number) -> Number`	#2
`func max(first : Number, second : Number) -> Number`	#3
`func max<T>(first : Number, second : T) -> Number`	#4
`func max<T>(first : T, second : Number) -> Number`	#4
`func max<T>(first : T, second : T) -> T`	#5

Tabella 4: Comparazione specificità di overload per la funzione `max`

Durante la fase di overload-resolution per un'ipotetica chiamata alla funzione `max` con argomenti `Int` e `Float`, allora vi sarebbero più overload compatibili, ma nell'ambito dei soli overload compatibili, solo uno di essi è nettamente più specifico di tutti gli altri, ovvero l'overload corrisponente alla seguente firma:

```
func max(first : Number, second : Number) -> Number
```

1.8.2 Funzioni extern

Una funzione esterna, in generale, è una funzione che non è definita all'interno della compilation unit in cui viene invocata. Nel caso concreto del linguaggio Basalt, essa è dunque una funzione che si desidera invocare all'interno di un programma Basalt, ma che è definita in un altro linguaggio ad esempio in C o in C++.

Il linker, in fase di linking, si occuperà di risolvere il riferimento alla funzione esterna a patto che esso abbia a disposizione l'object file contenente la definizione della funzione. Tale object file può essere emesso dai compilatori C e C++.

Per poter invocare una funzione esterna, è necessario che essa sia dichiarata all'interno del programma Basalt utilizzando la keyword *extern* al posto della keyword *func*, e rimpiazzando il corpo della funzione con un simbolo di **=** seguito da una string literal detta nome-macchina, rappresentante il nome con cui tale funzione viene nominata nell'object file che si desidera linkare, seguita da un punto e virgola.

Una funzione extern non può essere generica, e non vi possono essere più funzioni esterne associate allo stesso nome macchina.

```
// C stdlib: double sqrt(double);
extern square_root(value : Float) -> Float = "sqrt";
```

Tale design consente di poter effettuare overloading di funzioni esterne, in quanto il nome da utilizzare per l'overloading è quello con cui essa è stata dichiarata, che è potenzialmente differente dal nome che essa ha nell'object file, e che è stato invece specificato in coda alla dichiarazione. Di seguito un esempio di quanto detto:

```
// C stdlib: int32_t putchar(int32_t);
extern print(char : Char) = "putchar";

func print(str : RawString) {
    var index : Int = 0;
    while (str[index] != '\0') {
        print(str[index]);
        index = index + 1;
    }
}

func print(str : String) {
    var index : Int = 0;
    while (index < len(str)) {
        print(str[index]);
        index = index + 1;
    }
}
```

1.9 Immutabilità

Con il termine immutabilità si intende la proprietà di un oggetto di non poter essere modificato una volta creato. In Basalt, le variabili sono mutabili di default, ovvero è possibile modificarne il valore in qualsiasi momento. Tuttavia, esistono molteplici costrutti in basalt che permettono di ottenere l'immutabilità. Ciò che è immutabile non può essere modificato nè tramite assegnazione, nè tramite assegnazione ai suoi field, se è una struct, alle sue celle, se è una slice, un array o una stringa, nè all'area di memoria da esso, se è un puntatore.

1.9.1 Valori Letterali

Un valore letterale, in inglese "literal", sono quei valori che sono scritti direttamente nel codice sorgente. Ad esempio `42`, `true`, `"hello"` sono tutti valori letterali. In Basalt, i valori letterali sono immutabili, il che dovrebbe essere ciò che l'utente si aspetta.

1.9.2 Espressioni elementari

Un'espressione elementare è un'espressione che ricade in una delle seguenti categorie:

- Applicazione di un operatore binario (come ad esempio l'operatore di somma `+` o l'operatore di confronto `==`)
- Applicazione di un operatore unario non legato alla manipolazione di puntatori (come ad esempio l'operatore di negazione logica `!`)

Espressioni di questo tipo sono sostanzialmente paragonabili ai valori letterali, nel senso che esse sono da immaginarsi sostituibili dal loro risultato senza alcuna perdita di significato pertanto esse sono da considerarsi immutabili.

1.9.3 Espressioni di sola lettura

Un'espressione di sola lettura è un'espressione che restituisce un valore temporaneo preso per copia. Questo significa che anche qualora essa fosse mutabile, la modifica non sarebbe osservabile in alcun modo. Un esempio di espressione di sola lettura è la chiamata ad una funzione che restituisce un puntatore. Il valore restituito dalla funzione è un valore temporaneo, preso per copia, esso si dice essere in sola lettura e non può essere modificato. Tuttavia non è corretto parlare di immutabilità in quanto ciò che si trova all'area di memoria indirizzata dal puntatore potrebbe essere modificato.

```
func get_ptr() -> #Int { return memory::alloc<Int>(); }
var number : Int = 7;

// Errore di compilazione -> modifica di espressione di sola lettura
get_ptr() = &number;

// Ok -> modifica del valore puntato (non immutabile)
#get_ptr() = number;
```

1.9.4 Costanti

Una costante è nient'altro che una variabile immutabile. A differenza di linguaggi come Java e Kotlin, la garanzia di immutabilità per le costanti si estende non solo a loro stesse ma anche ai loro membri se sono struct, agli oggetti da loro puntati se sono puntatori e così via. In linea generale è possibile affermare che una costante non può essere modificata in nessuna sua parte. La dichiarazione di una costante necessita di un'inizializzazione, e avviene in accordo alla seguente sintassi: `const <nome> : <tipo> = <valore>;`

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

Ci si potrebbe chiedere quale sia l'utilità di un puntatore costante, dato che il puntatore stesso non può essere modificato così come il valore a cui punta. In effetti, l'utilità di un puntatore costante è limitata, specialmente se paragonata con quanto accade in C e C++, però la semantica della keyword `const` è molto differente. Un puntatore costante in Basalt può essere usato ad esempio come un alias, per tale scenario è riportato il seguente esempio:

```
struct Job {
    var name : String;
    var salary : Int;
}

struct Person {
    var name : String;
    var job : Job;
}

var person : Person = get_person();
    // Si assuma una funzione del genere esista

const job : #Job = &person.job;
```

In questo caso il puntatore `job` è costante, ciò implica che esso è immutabile e che non è possibile modificare nè il puntatore stesso nè ciò a cui punta utilizzando su di esso con operatore di dereferenziazione. Ciò non significa che ciò a cui il puntatore `job` punta sia destinato a non subire mai cambiamenti, in quanto accedendo in scrittura al campo `.job` della variabile `person`, la quale è mutabile, è possibile modificare il valore puntato da `job`.

In linguaggi come Java e Kotlin, le costanti (o `final` in Java) offrono garanzie di immutabilità solo per il riferimento, ma non per l'oggetto puntato. Questo significa che se si dichiara una costante di tipo `List`, è possibile modificare la lista aggiungendo o rimuovendo elementi, ma non è possibile assegnare un nuovo oggetto alla variabile costante. In Basalt, invece, una costante di tipo `List` non può essere modificata in nessun modo, nè aggiungendo nè rimuovendo elementi.

1.10 Assignments

Come in ogni linguaggio fortemente tipato, esistono regole ben precise che determinano quando è possibile assegnare un valore ad una variabile. In generale, è possibile assegnare espressioni non solo a variabili ma anche a vere e proprie espressioni.

Ci sono due tipi di vincoli che possono essere posti su un assegnamento: i vincoli di tipo ed i vincoli di immutabilità.

I vincoli di tipo sono il motivo stesso dell'esistenza di un linguaggio fortemente tipato, essi permettono di evitare errori di esecuzione dovuti ad un uso improprio delle variabili mediante un'analisi basata sul loro tipo dichiarato a tempo di compilazione.

I vincoli di immutabilità, invece, hanno a che fare con l'utilizzo delle costanti e/o delle espressioni immutabili (e.g. stringhe, numeri, ecc.).

In una assegnazione, in inglese "assignment", sono coinvolte due espressioni: l'espressione a sinistra dell'uguale (il target) e l'espressione a destra dell'uguale (il valore da assegnare). Ogni dichiarazione di costante o di variabile qualora inizializzata è implicitamente considerata un'assegnazione all'oggetto che si sta dichiarando.

1.10.1 Assignment semplici

Un assignment è ritenuto semplice se non coinvolge costanti o espressioni immutabili, e se il tipo del target non è nè un tipo Union, nè un tipo Array, nè un tipo Slice, nè un tipo Pointer, e se non ha parametri attuali di tipo.

Nelle condizioni appena descritte, l'assegnamento è possibile se e solo se i due nomi dei due tipi coinvolti corrispondono allo stesso identico tipo.

1.10.2 Assignment tra union

È utile immaginare una union come l'insieme dei tipi nominati direttamente o indirettamente al suo interno. È possibile assegnare ad una union espressioni il cui tipo è in tale insieme o espressioni il cui tipo è una union corrispondente ad un suo sottoinsieme.

Più formalmente, diciamo che è possibile assegnare ad espressioni di tipo union:

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

In altri termini, per le union e per le union soltanto si applica una politica di structural-compatibility, in luogo della name-equivalence.

1.10.3 Assignment tra puntatori

Quando il target di un'assignment è un puntatore scalare, è possibile assegnarvi solo espressioni che siano anch'essi puntatori scalari. Inoltre, è richiesto che i tipi degli oggetti puntati da entrambi coincidano strutturalmente, ovvero che essi siano identici oppure che essi siano union mutuamente assegnabili fra loro.

Ciò significa che anche se fosse possibile assegnare espressioni di tipo `V` a target di tipo `T`, qualora il viceversa non fosse vero, allora non sarebbe possibile assegnare espressioni di tipo `#V` a target di tipo `#T`.

Per quanto riguarda i puntatori vettoriali invece, vale quanto detto per i puntatori scalari ma con una dovuta precisazione, ovvero che è possibile assegnare ad un puntatore scalare `$T` un valore di tipo `[N]T`. Ciò è ovviamente vero in quanto in caso contrario verrebbe a mancare il senso stesso dei puntatori vettoriali, ovvero essere uno strumento per la gestione deglia array la cui dimensione è ignora a tempo di compilazione.

Per capire il motivo di tali restrizioni, è utile considerare il seguente esempio: si considerino due puntatori scalari `ptr : #Int` e `ptr2 : #(Int|Float)`, e si immagini cosa accadrebbe se fosse possibile assegnare `ptr` a `ptr2`. In tal caso, ci si ritroverebbe con un puntatore che punta ad un'area di memoria della dimensione sbagliata, il che potrebbe portare a comportamenti imprevedibili e a crash del programma, specialmente se si considera che tramite un riferimento a `ptr2` si potrebbe tentare di scrivere un valore di tipo `Float` in un'area di memoria che può contenere solo valori di tipo `Int`.

1.10.4 Assignment tra tipi generici

Considerando un qualunque tipo generico `Example`, qualora il target di un assignment fosse un tipo generico `Example<`T_1`, `T_2`, ...>`, vale la pena sottolineare che è possibile assegnare ad esso espressioni il cui tipo è `Example<`U_1`, `U_2`, ...>`, se e solo se i tipi T_i e U_i coincidono strutturalmente, ovvero se e solo se i tipi T_i e U_i sono mutuamente assegnabili fra loro.

1.10.5 Assignment Tra Array

Gli assignment a target di tipo array sono possibili solo se il tipo del valore da assegnare è anch'esso un array della medesima dimensione.

Essendo possibile vedere degli assignment tra array (e non slice), come una sequenza di assignment membro a membro, valgono le stesse regole che sono state discusse fin ora.

L'assegnazione di un array ad un altro array è un'operazione che potrebbe essere effettuata in tempo lineare qualora tali array contengano espressioni di tipo union e/o qualora l'architettura per cui si desidera compilare non supporti istruzioni SIMD (Single Instruction Multiple Data).

1.10.6 Assignment verso target immutabili

Assegnare un valore ad un target immutabile, ovvero ad un target costante o letterale (e.g. stringhe, numeri, ecc.), è proibito, e porta ad errori a tempo di compilazione.

É opportuno sottolineare che, in generale, un target è costante e dunque immutabile anche se esso è un membro di una costante di tipo struct, l'oggetto puntato da un puntatore costante, una cella di una slice costante o un elemento di un array costante

Il risultato di un'operazione binaria è sempre immutabile (ad esempio la somma di due numeri è immutabile) mentre il risultato di un'operazione unaria è immutabile solo se l'operando è immutabile in tutti i casi eccetto per l'operatore di dereferenziazione di un puntatore, il quale restituisce un valore mutabile per puntatori non costanti.

1.10.7 Assignment di espressioni immutabili

Assegnare espressioni immutabili a target immutabili è sempre permesso (ciò per costruzione può solo avvenire in sede di dichiarazione di una costante), assegnarli a target mutabili invece, è permesso solo se tale assegnamento non implica un legame del valore immutabile con un target mutabile.

Un legame di un valore immutabile con un target mutabile si ha ad esempio provando ad assegnare a tale target l'indirizzo di memoria del valore immutabile in formato di puntatore scalare o vettoriale.

Tale legame consentirebbe infatti di modificare il valore immutabile deferenziando il puntatore mutabile così ottenuto, il che è chiaramente in contrasto con la natura stessa del concetto di immutabilità.

1.10.8 Assignment verso espressioni di sola lettura

Un concetto affine a quello di immutabilità è quello di sola lettura. Un'espressione è considerata di sola lettura se essa è un valore mutabile ma il cui potenziale mutamento non porterebbe alcun effetto visibile all'esecuzione del programma in nessuna circostanza. Ad esempio, con il seguente frammento di codice si vuole mostrare una selezione di assignment, alcuni dei quali non validi in quanto tentano di assegnare un valore mutabile ad un target che in teoria sarebbe mutabile, ma che nel contesto di riferiemento è considerato immutabile in quanto il suo mutamento non porterebbe alcun effetto visibile all'esecuzione del programma.

```
get_pointer_to_int() = &x; // Errore
get_array_of_ints()[0] = 42; // Errore
get_slice_of_ints()[0] = 42; // Ok
get_pointer_to_struct().field = 42; // Ok
var x : Int = #get_pointer_to_int(); // Ok
```

1.11 Pseduo-polimorfismo

Con il termine polimorfismo in informatica, ci si riferisce alla capacità di un linguaggio di poter astrarre dal tipo concreto di un oggetto, permettendo di scrivere codice che possa essere applicato a tipi diversi, i quali condividono la stessa API.

Questo concetto viene spesso legato a doppio filo con quello di ereditarietà nei contesti di programmazione ad oggetti, in quanto l'API condivisa a tutti i tipi concreti su cui si desidera operare viene codificata nella forma della loro classe base (parent-class).

In Basalt, così come in Go, Rust e altri linguaggi moderni, è possibile ottenere gli stessi effetti del polimorfismo object-oriented senza dover ricorrere all'ereditarietà. In particolare Basalt utilizza un approccio unico nel panorama dei linguaggi di programmazione di basso livello, che sfrutta una feature chiamata *Common Features Adoption (CFA)* per implementare così una forma di pseudo-polimorfismo.

1.11.1 Common features adoption (CFA)

Con *Common Features Adoption*, abbreviato come CFA, ci si riferisce all'abilità del compilatore di generare un'overload di una funzione, basandosi sugli overload già esistenti, direttamente in sede di chiamata. In particolare, gli overload auto-generati implicitamente tramite CFA sono costruiti in modo tale da effettuare un dispatch a tempo di esecuzione sui tipi concreti degli argomenti di una chiamata a funzione, in modo da poter chiamare il corretto overload già esistente.

Si considerino ad esempio le seguenti definizioni di funzioni:

```
func half(x : Int) -> Int {
    return x / 2;
}

func half(x : Float) -> Float {
    return x * 0.5;
}
```

Se si effettuasse una chiamata alla funzione `half`, con un argomento di tipo `Int|Float`, il sistema non troverebbe un overload specifico. Esso però sarebbe in grado di generare un overload ad-hoc dietro le quinte simile a quanto riportato di seguito:

```
func half(x : Int | Float) -> Int | Float {
    if (x is Int) {
        return half(x as Int);
    } else {
        return half(x as Float);
    }
}
```

Tutto ciò che il programmatore dovrà fare sarà effettuare una chiamata a funzione. Qualora non esistesse un overload specifico per i tipi degli argomenti passati, il compilatore proverà a generare un overload definito per casi analizzando uno per uno tutti gli argomenti passati in chiamata da sinistra a destra.
Si considerino infatti i seguenti overload per la funzione `add`:

```
func add(x : Int, y : Int) -> Int {
    return x + y;
}

func add(x : Float, y : Int) -> Float {
    return x + utils::convert_to<Float>(y);
}

func add(x : Int, y : Float) -> Float {
    return utils::convert_to<Float>(x) + y;
}

func add(x : Float, y : Float) -> Float {
    return x + y;
}
```

In questo contesto, sarebbe possibile effettuare le seguenti chiamate a funzione:

CHIAMATA A FUNZIONE	***DEFINIZIONE CORRISPONDENTE***
`add(Int, Int)`	overload definito dall'utente
`add(Int, Float)`	overload definito dall'utente
`add(Float, Int)`	overload definito dall'utente
`add(Float, Float)`	overload definito dall'utente
`add(Int|Float, Int)`	overload generato tramite CFA
`add(Int|Float, Float)`	overload generato tramite CFA
`add(Int, Int|Float)`	overload generato tramite CFA
`add(Float, Int|Float)`	overload generato tramite CFA
`add(Int|Float, Int|Float)`	overload generato tramite CFA

Tabella 5: Comparazione specificità di overload per la funzione `add`

1.11.2 Implicazioni della CFA

Considerato quanto detto finora, è possibile trarre alcune conclusioni riguardo i benefici e le limitazioni che derivano dall'utilizzo della CFA in Basalt.

È necessario sottolineare come la CFA consenta di esporre al programmatore un'API basata sulle funzionalità comuni a tutti i tipi concreti che è possibile assegnare a un tipo base. Nel caso di Basalt, tale tipo base è una union. Questo significa che il programmatore può scrivere codice che opera su un tipo base, senza dover conoscere il tipo concreto, usando le funzionalità comuni a tutti i tipi concreti proprio come se tale tipo base fosse un'interfaccia di un linguaggio ad oggetti (e.g. Java, Kotlin, C#).

Ad esempio, sarà possibile chiamare la funzione `size` su una union dei tipi `List<T>`, `Tree<T>`, `HashSet<T>` come se tale union fosse un'interfaccia che esponesse tale funzionalità. Affinché ciò avvenga, sarà necessario che esista un overload definito dall'utente della funzione `size` per tutti i tipi citati.

1.11.3 Considerazioni e compromessi riguardanti la CFA

Risulta evidente come la CFA possa risultare peggiore dell'ereditarietà in termini di performance, in quanto ogni singola chiamata a funzione potrebbe comportare un controllo in tempo lineare sul numero di tipi concreti che è possibile assegnare ai vari tipi base degli argomenti.

Ciò nonostante, la CFA consente di scrivere codice estremamente flessibile e modulare, senza costringere il sistema a dover tener traccia di v-tables e/o object-headers di sorta, i quali sono invece necessari per implementare il polimorfismo tramite ereditarietà come ad esempio accade in Java, Kotlin, C#, i quali sono costi di overhead che impattano l'intera codebase, e non solo le sezioni che necessitano di usare il polimorfismo.

Basalt è ottimizzato per le situazioni dove non è necessario polimorfismo dinamico, in quanto ci si aspetta che solo una minima parte della codebase necessiti di tale feature, e pertanto, si ritiene che sia più giusto pagare un costo anche considerevole in termini di performance in situazioni mediamente rare pur di ottenere codice meglio performante in tutti gli altri scenari.

Si tenga poi a mente che per scenari dove il numero totale di tipi concreti da considerare per la generazione di overload CFA è molto contenuto (non più di 5 tipi), la CFA potrebbe risultare competitiva in termini di performance dato che il numero totale di istruzioni macchina corrispondenti a risolvere un riferimento a metodo in una v-table non è troppo dissimile dal numero di istruzioni necessarie per risolvere tale overload CFA.

2 Implementazione

2.1 Generalità sul processo di compilazione

Il processo di compilazione, per sua natura, è un processo sequenziale schematizzabile come una pipeline (catena di montaggio). Ogni fase della compilazione ha una responsabilità ben definita e produce un output che sarà l'input della fase successiva.

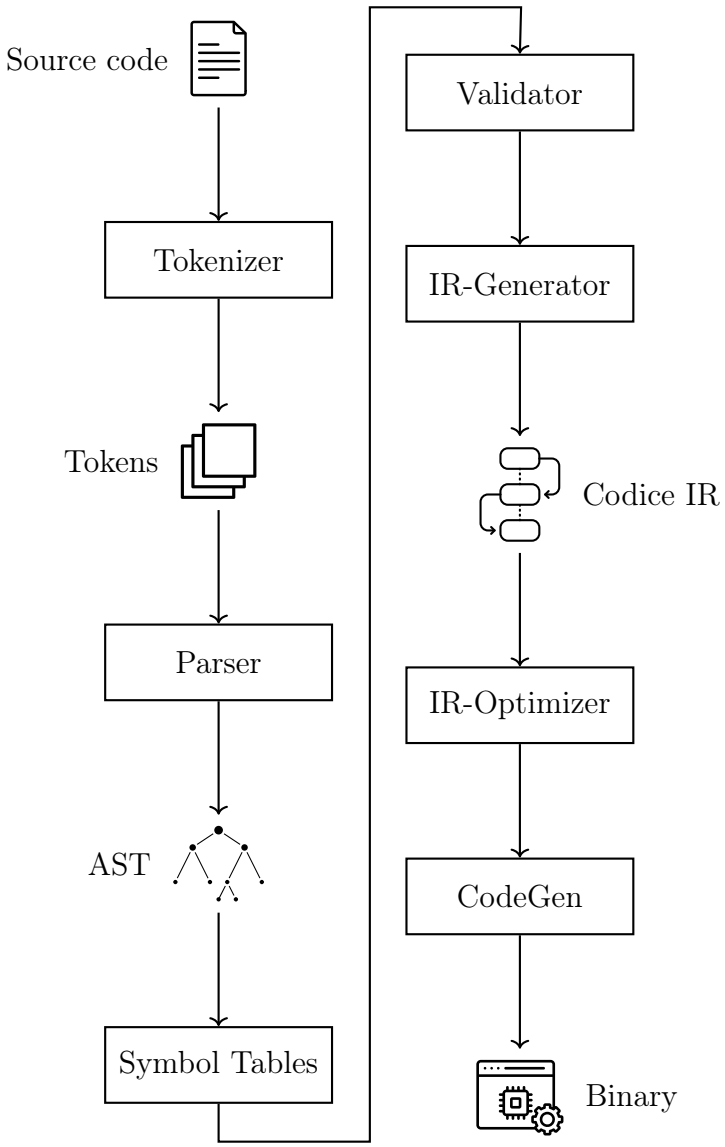


Figura 6: Pipeline del processo di compilazione

2.1.1 Tokenizzazione

Con tokenizzazione, si intende il processo di suddivisione del codice sorgente in *token*, ovvero in unità atomiche che rappresentano i componenti del linguaggio. Queste unità atomiche possono essere rappresentate come semplici stringhe, oppure come entità più ricche di informazioni, come ad esempio il nome del file sorgente dal quale sono stati estratti, la posizione all'interno del file (riga, colonna) e così via.

Il processo di tokenizzazione è il primo passo del processo di compilazione, ed è possibile implementare un tokenizzatore in diversi modi. In generale, è utile immaginare un tokenizzatore come un algoritmo iterativo che dato un input testuale continua a leggere caratteri finché essi non formano un token valido. Una volta che un token è stato riconosciuto, esso viene conservato in un'opportuna collezione.

Commenti, spazi e alcuni caratteri speciali sono generalmente ignorati dal tokenizzatore, nel senso che essi vengono correttamente riconosciuti ma non vengono conservati nella collezione.

Un token che non viene riconosciuto porta ad un errore a tempo di compilazione.

2.1.2 Parsing

Con *Abstract Syntax Tree* (AST) si intende una struttura dati ad albero che rappresenta un'espressione, uno statement o una definizione di una variabile in un linguaggio di programmazione. L'AST è in corrispondenza biunivoca con il codice sorgente, e viene utilizzato per rappresentare il codice sorgente in una forma più adatta per l'analisi e la manipolazione da parte del compilatore.

L'atto di trasformare una collezione ordinata di token in un AST **è chiamato *parsing***.

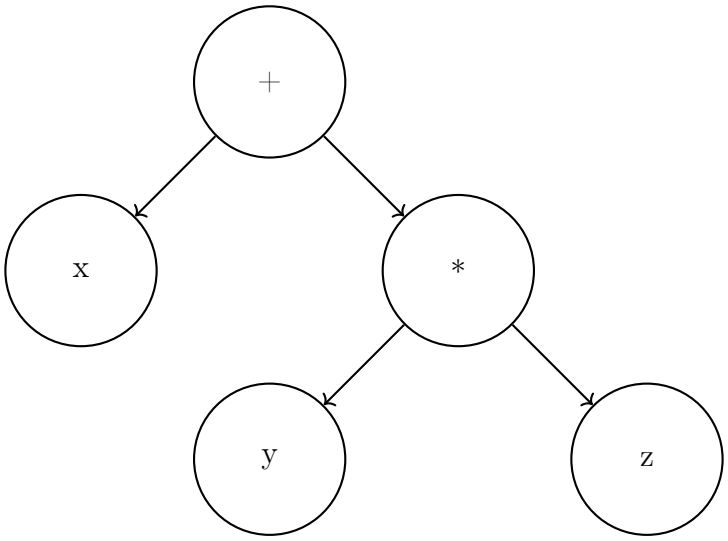


Figura 7: Esempio di generico AST per un'espressione aritmetica "x + y * z"

In pratica, il compilatore Basalt utilizza internamente AST simili al seguente. Tale albero è ben più strutturato in quanto ogni nodo rappresenta un'entità del linguaggio ben precisa.

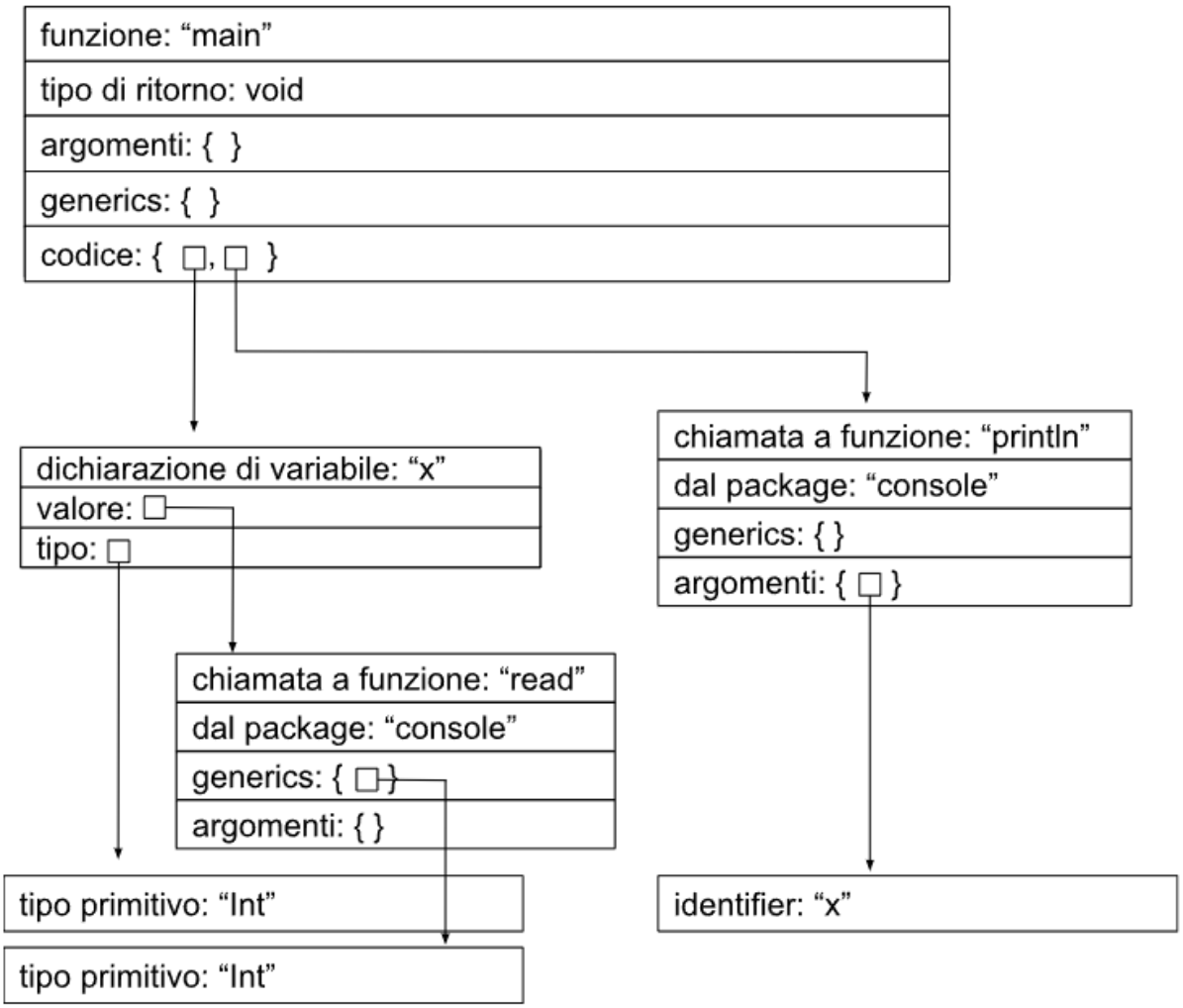


Figura 8: Esempio di AST per una funzione main che legge da riga di comando un numero intero e lo stampa subito dopo

Nei capitoli successivi sarà reso conto di come sia stato implementato il parsing in Basalt in dettaglio. Per completezza, si riporta che esistono due principali famiglie di algoritmi di parsing: i parser *LL* ed i parser *LR*. Tali parser differiscono per il modo in cui essi costruiscono l'AST. I parser LL costruiscono l'AST scansionando i token da sinistra a destra e costruendo il sottoalbero sinistro completamente prima di passare al token seguente (leftmost-derivation). I parser LR, invece, seppur scansionano i token da sinistra a destra, costruiscono l'AST modificando e rifinendo il sottoalbero destro man mano che scoprono nuovi token (rightmost-derivation).

2.1.3 Costruzione delle symbol-table

Le symbol-table (tabelle dei simboli), sono strutture dati che memorizzano le varie definizioni di funzioni e tipi presenti all'interno del programma in un formato che ne facilita il recupero.

Sostanzialmente si tratta di strutture dati chiave-valore in cui ad ogni chiave, che spesso è un AST, ad esempio relativo ad una chiamata a funzione o ad una type-signature, viene associata una definizione, anch'essa nella forma di AST, relativo alla definizione corrispondente.

Tipicamente esse sono costruite come wrapper su strutture dati più semplici, come ad esempio delle hash-map, ed implementano internamente una traduzione da AST (chiave per la symbol table) a stringa (chiave per la hash-map), con eventuali meccanismi di caching più o meno sofisticati per evitare di dover ricalcolare la chiave dell'hash-map ad ogni accesso.

2.1.4 Validazione ed analisi statica

La fase di validazione è una delle fasi più importanti del processo di compilazione. Essa consiste nel navigare con uno o più visitor l'AST generato dalla fase di parsing e verificare che esso sia corretto rispetto a determinate regole semantiche.

Basalt, effettua i seguenti controlli di validità sul codice sorgente:

- Aciclicità delle dipendenze dirette tra tipi: (Non esistenza di struct o union definite per ricorsione diretta)
- Non ambiguità dei tipi (Non esistenza di tipi con lo stesso nome e con lo stesso numero di parametri formali di tipo nello stesso package, o in package diversi ma importati in uno stesso file)
- Address sanitizing (Verifica che non si stia provando a calcolare l'indirizzo di un entità non allocata)
- Typechecking (Verifica che tutti i tipi usati esistano e siano coerenti con il contesto, controllo degli assignments per correttezza di tipo, risoluzione delle chiamate a funzione e controllo sui tipi dei parametri, sui generics e sul tipo di ritorno)
- Immutability-checking (Ispezione degli assignments e delle chiamate a funzioni ia fini di impedire modifiche a entità immutabili quali costanti e/o espressioni di sola lettura)
- Exit-path-checking (Ispezione dei flussi di esecuzione delle funzioni, ai fini di garantire l'assenza di codice irraggiungibile e la presenza di un return statement in tutti i possibili flussi di esecuzione per funzioni non void)

2.1.5 Conversione dell'AST in IR

Una volta che tutte le definizioni, in forma di AST, sono state validate, esse vengono convertite in IR (Intermediate Representation). Questa nuova forma di rappresentazione del codice sorgente permette di eseguire operazioni di ottimizzazione e di generazione del codice macchina in modo più efficiente.

Il codice macchina infatti è per sua natura lineare, ovvero è una sequenza ordinata di istruzioni. Il processore possiede un registro chiamato Program-Counter (PC) il quale tiene traccia dell'indirizzo dell'istruzione corrente, al termine di ogni istruzione il PC viene incrementato o decrementato in modo da puntare all'istruzione successiva.

La sfida della fase di traduzione dell'AST in IR consiste nel trasformare una struttura ad albero in una sequenziale, utilizzando le istruzioni di salto per rispecchiare fedelmente il flusso di esecuzione atteso.

Si consideri ad esempio il seguente frammento di codice, rappresentabile in forma di AST come mostrato in Figura 10 (si propone una rappresentazione semplificata):

```
if (cond1) {
    if (cond2) {
        console::println("cond1 && cond2");
    }
    else {
        console::println("cond1 && !cond2");
    }
}
else {
    console::println("!cond1");
}
```

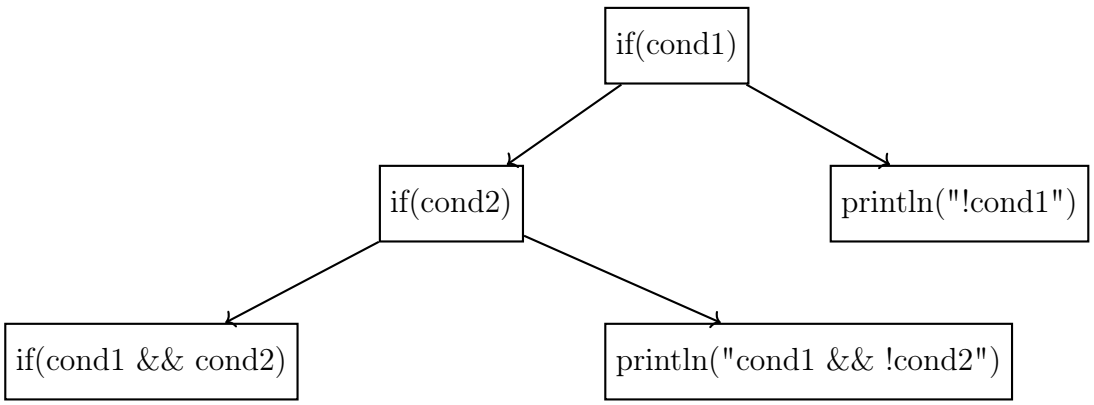


Figura 9: Esempio di AST **semplificato** per un doppio if-else annidato

L'AST nella figura precedente rappresenta un doppio if-else annidato. La conversione in IR di tale AST richiede l'utilizzo di istruzioni di salto condizionato per gestire correttamente il flusso di esecuzione. (anch'esso semplificato per motivi di leggibilità)

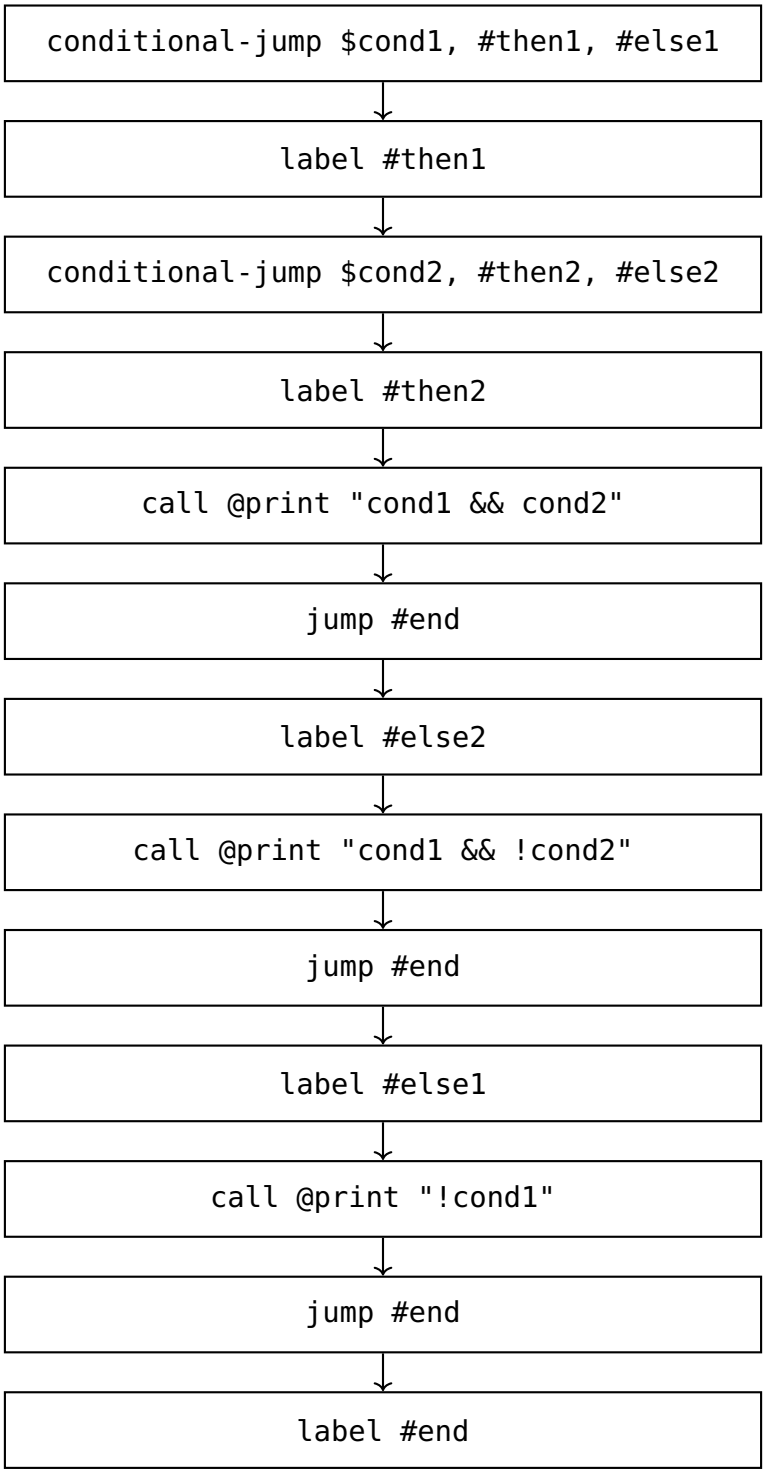


Figura 10: Esempio di IR **semplificato** per un doppio if-else annidato

2.1.6 Ottimizzazione dell'IR

Una volta che l'IR è stato generato, esso si assume semanticamente corretto, e si può finalmente perdere traccia dell'AST, delle symbol-table e di tutte le altre strutture dati intermedie, comprese tutte le informazioni relative alle coordinate dei vari elementi del programma all'interno del sorgente.

La fase di ottimizzazione dell'IR consiste nel migliorare il codice IR mediante l'applicazione di trasformazioni che ne riducano la complessità e ne migliorino le prestazioni ma che ne lasciano invariati gli effetti osservabili.

Le ottimizzazioni possono essere di vario tipo, alcune delle più comuni sono:

- **Constant folding**: Calcolo di espressioni costanti in fase di compilazione.
- **Common subexpression elimination**: Rimozione di espressioni comuni
- **Loop minimization**: Estrazione di codice dal corpo dei cicli iterativi
- **Inlining**: Sostituzione di chiamate a funzione con il corpo della funzione stessa
- **Register optimization**: Utilizzo più efficiente dei registri del processore
- **Code Reordering**: Riordinamento di istruzioni non dipendenti fra loro per poter raggruppare istruzioni che possono essere eseguite in parallelo in un'unica sezione
- **SIMD**: Utilizzo di istruzioni SIMD (Single Instruction Multiple Data) al posto di sequenze di istruzioni normali ripetute (Possibile solo per alcune architetture)

Una trattazione dettagliata dell'argomento è fuori dallo scopo di questo documento, in quanto è un ambito estremamente ampio ed in continua espansione.

2.1.7 Conversione dell'IR in codice macchina

Il codice IR, una volta ottimizzato, viene convertito in codice macchina mediante un processo chiamato *code-generation*. Il codice IR per sua natura è estremamente simile al codice macchina in struttura e semantica, e quindi la conversione è un processo relativamente semplice.

Ogni istruzione IR è direttamente rimpiazzabile con una o più istruzioni macchina che eseguono la stessa operazione. A differenza del codice IR, che è cross-platform, il codice macchina è specifico per l'architettura del processore su cui si intende eseguire il programma.

Le istruzioni macchina sono codificate in linguaggio binario, e sono scritte su file in modo incrementale man mano che vengono generate. Il file risultante è un file oggetto che è possibile linkare ed eseguire.

2.2 Frontend/backend compiler-frameworks

Adottando una terminologia diffusasi primariamente nel contesto della programmazione web, è possibile introdurre i concetti di frontend e backend in un compilatore. Con il frontend di un compilatore si intende l'insieme di componenti software (package, classi, metodi, funzioni) che si occupano di implementare le fasi di tokenizzazione e di parsing del codice sorgente. Con il termine backend, invece, si fa riferimento all'insieme di componenti software che si occupano di implementare le fasi di ottimizzazione dell'IR e di codegen.

Con il passare del tempo, ci si è accorti che estrapolare dalle codebase dei vari compilatori i loro frontend e backend, per poi generalizzarli e renderli riutilizzabili, avrebbe portato a una maggiore efficienza nello sviluppo di nuovi linguaggi. Questo ha portato alla nascita di frameworks per la creazione di compilatori, che offrono funzionalità più o meno avanzate negli ambiti del frontend e del backend, in modo da lasciare liberi i designer di un dato compilatore di concentrarsi sull'analisi statica.

Questo ha consentito, ad esempio, a linguaggi come Rust di svilupparsi in termini di features semantiche, quali ad esempio il borrow checker, senza dover preoccuparsi prima di implementare un backend perfettamente funzionante da zero.

2.2.1 LLVM: Low Level Virtual Machine

Ciò che accomuna Basalt, Rust, Zig e molti altri linguaggi moderni è l'adozione di LLVM come backend-framework. LLVM è un progetto open-source che si occupa di offrire una API per costruire manualmente rappresentazioni IR per le varie istruzioni di un programma, e di fornire un set di tool per ottimizzare e generare codice macchina a partire da tali rappresentazioni IR.

In altri termini, LLVM si occupa di implementare le fasi di IR-optimization e codegen discusse in precedenza, essendo cross-platform, ovvero supportando varie CPU e varie architetture (x86, ARM, Web-Assembly, etc...).

il team di LLVM supporta ufficialmente C e C++, ma vari porting non ufficiali esistono per molteplici altri linguaggi.

LLVM ha infine reso possibile la creazione di cling, un interprete C/C++ basato sul compilatore clang, il quale utilizza LLVM per effettuare JIT-compilation (compilazione "just in time", ovvero al volo) di codice C/C++. Tale strumento è tutt'ora di fondamentale importanza per la comunità scientifica, ed è in utilizzo presso il CERN.

2.2.2 Introduzione ad LL (LLVM-IR)

LLVM, come già detto nel paragrafo precedente, è un progetto composto da un insieme di librerie, e da alcuni tool. Due di questi tool sono `lli` ed `llc`, rispettivamente un interprete IR e un compilatore IR.

Questi due strumenti possono essere usati per eseguire codice IR, fornito sotto forma di file di testo con estensione `.ll`. Analizzare la sintassi e, più in generale, la semantica di tale formato testuale renderà più facile comprendere l'utilizzo della API nei capitoli successivi.

Un simbolo globale deve essere preceduto dal carattere speciale "@", ad esempio la funzione che fa da entry-point per l'esecuzione del programma è definita come `@main`.

Per definire una funzione è necessario specificare il tipo di ritorno, il nome della funzione, e la lista dei parametri. Il corpo della funzione è racchiuso tra parentesi graffe.

```
define i32 @main() {
  entry:
    ret i32 0
}
```

Una funzione in LL è fatta da uno o più blocchi, ognuno dei quali è composto da una lista di istruzioni. Ogni blocco termina con un'istruzione di return o di salto. Nella funzione `@main` definita sopra, `entry` è una label, e definisce l'ingresso nell'omonimo blocco. Il blocco entry è un blocco speciale in quanto esso è il primo ad essere eseguito in ogni funzione.

L'istruzione `ret` effettua il return e, pertanto, sancisce la fine del blocco corrente. Il valore restituito deve essere dello stesso tipo specificato nella firma della funzione e tale tipo va esplicitamente specificato.

I nomi dei simboli possono contenere spazi e/o caratteri speciali purchè racchiusi tra doppi apici. Per effettuare un return da una funzione void servirà scrivere `ret void`.

```
define void %"example void function"() {
  entry:
    ret void
}
```

L'utilizzo di "%" in luogo di "@" identifica un simbolo come locale invece che globale, pertanto esso non sarà visibile all'esterno del file `.ll` in cui esso è definito.

2.2.3 Implementazione di strutture dati in LL

È possibile definire delle struct in LL, le quali sono simboli come le funzioni e possono essere globali o locali. Di seguito è riportato un esempio di definizione di due struct locali, che rappresentano rispettivamente uno stack (struttura dati) ed un suo nodo.

```
%Stack = type {
    %Node*,
    i64
}

%Node = type {
    %Node*,
    i32
}
```

Volendo mostrare solo un esempio di come è possibile usare le struct definite sopra, si riporta la definizione di una funzione che stampa tutti gli elementi di uno stack.

Nel seguente frammento di codice `.ll` si può notare che, per chiamare una funzione, si utilizza l'istruzione `call`, la quale richiede l'esplicita specificazione del tipo di ritorno della funzione chiamata. L'istruzione `br` è una istruzione di salto.

```
define void @printStack(%Stack* %stack) {
  entry:
    %cursorPtr = call %Node* @getHead(%Stack* %stack)
    br label %loopCondition

  loopCondition:
    %cursor = load %Node*, %Node** %cursorPtr
    %condition = icmp ne %Node* %cursor, null
    br i1 %condition, label %loopBody, label %loopExit

  loopBody:
    %currentNumberPtr = call i32* @getNumber(%Node* %cursor)
    %currentNumber = load i32, i32* %currentNumberPtr
    call void @printInt(i32 %currentNumber)
    %currentNextPtr = call %Node* @getNext(%Node* %cursor)
    %currentNext = load %Node*, %Node** %currentNextPtr
    store %Node* %currentNext, %Node** %cursorPtr
    br label %loopCondition

  loopExit:
    ret void
}
```

2.2.4 Utilizzo della memoria in LL

In LL, la memoria è gestita in modo drasticamente diverso rispetto a come avviene nei linguaggi di programmazione di uso comune.

Nei linguaggi di programmazione tipici una variabile è associata ad un indirizzo di memoria ed è possibile assegnare valori a tale variabile riferendosi ad essa con il suo nome. Quando servirà leggere il valore di tale variabile, sarà sufficiente, ancora una volta, riferirsi ad essa con il suo nome.

In LL ciò non avviene, in quanto le variabili non hanno un vero e proprio indirizzo di memoria e sono invece solo dei simboli. Si consideri ad esempio il seguente frammento di codice Basalt e la sua fedele traduzione in LL:

```
var number : Int = 42;
console::println(number);
```

```
example_block:
  %numberAddress = alloca i64
  store i64 42, i64* %numberAddress
  %numberValue = load i64, i64* %numberAddress
  call void @"console::println<Int>"(i64 %numberValue)
```

Si noti come la variabile `number` sia stata tradotta come la `numberAddress` e `numberValue`. La variabile `numberAddress` è un puntatore ad una locazione di memoria e la variabile `numberValue` è il valore numerico contenuto in tale area di memoria.

Nè `numberValue` nè `numberAddress` in questo caso hanno dei veri e propri indirizzi di memoria ad essi associati. Essi sono solo dei simboli a cui sono associati dei valori.

La gestione della memoria di LL può sembrare controintuitiva al programmatore medio, ed è proprio per questo che anche se LL è un vero e proprio linguaggio, esso non è considerato veramente utilizzabile. Si è infatti abituati a ragionare in termini di variabili, intese come coppie indirizzo/valore, ed un approccio simile è facilmente classificabile come innaturale.

È bene ricordarsi però che i calcolatori utilizzano internamente una rappresentazione analoga a quella proposta da LL ed è proprio per questo che LL è facilmente ed efficientemente traducibile in linguaggio macchina nativo per molteplici architetture.

2.2.5 ANTLR: Another Tool for Language Recognition

ANTLR è un progetto open-source che si occupa di offrire uno strumento per generare codice in vari linguaggi, a partire da grammatiche e vocabolari in formato g4, che rappresentano la sintassi di un linguaggio di programmazione. Il codice generato da ANTLR è difatti un'intera unità di frontend provvista di tokenizer (lexer), parser e anche di un visitor, ovvero una classe astratta che può essere estesa per implementare agevolmente meccanismi di navigazione dell'AST.

Assieme al codice autogenerato, è necessario aggiungere al progetto una dipendenza, ovvero il runtime di ANTLR, che si occupa sostanzialmente di fornire le funzionalità di base sulle quali il codice autogenerato si basa.

2.2.6 ANTLR: Vocabolari e grammatiche

ANTLR lavora con due file di testo in formato `.g4`, che rappresentano rispettivamente il vocabolario e la grammatica del linguaggio.

Un vocabolario è un file contentente le regole di tokenizzazione in forma di espressioni regolari. Ogni regola è composta da un nome, seguito dai due punti, e da un'espressione regolare oppure un exact-match delimitato da apici.

Il seguente frammento di vocabolario `.g4` è un'estratto dal vocabolario di Basalt, dove si definiscono i token `ID` e `TYPENAME`, rispettivamente per gli identificatori (nomi di variabili, costanti, funzioni) e per i nomi dei tipi.

```
ID       : [a-z][a-zA-Z_0-9]* ;
TYPENAME : [A-Z][a-zA-Z_0-9]* ;
```

Una grammatica è un file contentente le regole di parsing del linguaggio. Ogni regola è composta da un nome, seguito dai due punti, e da una sequenza di token e/o regole eventualmente combinate.

Il seguente frammento di grammatica `.g4` è un'estratto dalla grammatica di Basalt, dove si definiscono le regole per il parsing di variabili e costanti.

```
variableDeclaration
    : VAR ID COLON typeSignature ASSIGN expr SEMICOLON
    | VAR ID COLON typeSignature SEMICOLON
    ;

constDeclaration
    : CONST ID COLON typeSignature ASSIGN expr SEMICOLON
    ;
```

2.2.7 ANTLR: Generazione del frontend

Una volta spostatosi nella directory dove sono presenti i file `.g4`, è possibile generare il frontend da riga di comando utilizzando l'apposito cli-tool di ANTLR, ovvero `antlr4`.

Per generare il frontend, è necessario specificare il linguaggio di destinazione, il quale può essere Java, Python, C++, ecc...

```
antlr4 -Dlanguage=Cpp -no-listener -visitor *.g4
```

2.2.8 Considerazioni generali

L'utilizzo di frameworks nello sviluppo di compilatori è una pratica ormai consolidata, anche se vale la pena chiedersi se tale pratica sia sempre conveniente.

LLVM è un framework largamente usato da numerosi compilatori per linguaggi di successo, diffusi, performanti, sicuri ed efficienti. LLVM ha dimostrato di essere un framework affidabile e robusto, e soprattutto la sua larga user-base ne garantisce la manutenzione e l'aggiornamento costante, il quale a sua volta garantisce la compatibilità con le ultime versioni di CPU e architetture, che in un periodo come questo è molto importante data la proliferazione di nuove archietture ARM / RISC-V.

L'egemonia dell'architettura x86 sarà, con ogni probabilità, messa in discussione nei prossimi decenni, pertanto lo sviluppo di un backend da zero, richiederebbe uno studio approfondito di molti standard, diversi fra loro, ed in continuo aggiornamento.

In definitiva, l'utilizzo di LLVM come backend-framework è una scelta proficua, che permette di concentrarsi su ciò che realmente conta, ovvero le symbol-tables e l'analisi statica.

Per quanto ANTLR invece, la situazione è un po' diversa. ANTLR è un framework molto potente, ma vale la pena sottolineare che esso, occupandosi di frontend a tutto tondo, si occupa anche della gestione dei commenti e nell'emissione degli errori di sintassi. Questo comporta un minor controllo sul tipo di feedback da dare all'utente, il quale potrebbe ricevere messaggi di errore non chiari o addirittura fuorvianti.

Inoltre, utilizzare ANTLR in C++, porta a dei problemi di compatibilità per via dell'utilizzo di alcune features deprecate dei vecchi standard C++ all'interno del codice autogenerato. Ciò è facilmente risolvibile mediante correzione manuale, ma ciò andrebbe a inficiare la facilità di manutenzione e di aggiornamento, rendendo difficile modificare la grammatica del linguaggio (ogni modifica della grammatica richiederebbe una nuova generazione e dunque una nuova correzione manuale).

2.3 Sviluppo del compilatore Basalt

Alla luce di quanto detto nei capitoli precedenti, si è deciso di sviluppare il compilatore per il linguaggio di programmazione Basalt in C++. Questa scelta è stata fatta per diversi motivi, tra cui:

- **Facilità di utilizzo del compilatore**: É stato dato un particolare peso alla facilità di utilizzo del prodotto finito. Un compilatore scritto in Java ad esempio, avrebbe richiesto l'installazione di una JVM, rendendo il prodotto meno accessibile (Considerazioni analoghe possono essere fatte per C# o simili). Inoltre, in C++ è possibile creare eseguibili linkati staticamente, semplificando il processo di distribuzione del compilatore
- **Performance**: C++ è un linguaggio altamente performante, requisito fondamentale per un compilatore. Un compilatore scritto in un linguaggio più lento avrebbe richiesto tempi di compilazione più lunghi, rendendo l'esperienza dell'utente finale meno piacevole
- **Supporto di LLVM**: L'adozione di LLVM è stata una scelta compiuta fin dalle primissime fasi di design del compilatore. La pool di linguaggi supportati da LLVM è comunque piuttosto ristretta, rendendo C++ una scelta quasi obbligata. Inoltre, LLVM è scritto in C++, rendendo la scelta ancora più naturale.

2.3.1 Doppia repository: Con e senza ANTLR

Il progetto Basalt è stato suddiviso in due repository distinte. Una di esse, denominata *Basalt*, contiene il compilatore vero e proprio, scritto in C++ e basato su LLVM. L'altra, denominata *unina-Basalt*, contiene l'adattamento della codebase principale ad ANTLR4.

Sono state create due repository per toccare con mano i vantaggi e gli svantaggi dell'utilizzo di ANTLR, i quali sono stati già discussi in precedenza. É opportuno evidenziare come se ANTLR non causasse problemi di compatibiltà che inficiano la riproducibiltà delle build, la repository basata su ANTLR avrebbe rimpiazzato la repository principale.

Repository	***URL***
`unina-Basalt`: basata su ANTLR4	`www.github.com/fDero/unina-Basalt`
`Basalt`: principale	`www.github.com/fDero/Basalt`

Tabella 6: Repository github

2.3.2 Build automatizzata

Posto che la build del compilatore nella sua versione basata su ANTLR è manuale, e può essere condotta solo usando specificamente il compilatore gcc con lo standard 17, la build del compilatore Basalt nella sua versione proposta sulla repository principale è automatizzata.

La build automatizzata utilizza conan, un package manager per C++, per scaricare le dipendenze del progetto (LLVM e libxml2), le quali sono poi compilate in locale automaticamente solo al primo utilizzo.

Scaricate le dipendenze, il progetto viene compilato con cmake, che è uno strumento che consente di effettuare build incrementali di progetti C e C++.

2.3.3 Installer per Windows x86

Per facilitare l'installazione del compilatore Basalt su sistemi Windows x86, è stato creato un installer automatico nel formato msi (Microsoft Installer).

Tale installer posiziona l'eseguibile (staticamente linkato) del compilatore Basalt nella directory `%Program Files%\basalt\<version>`, e aggiunge la directory alla variabile d'ambiente `PATH`. In fase di disinstallazione, che avviene dal pannello di controllo, rimuove l'eseguibile e la directory dalla variabile d'ambiente.

Tale installer è stato creato con WiX Toolset, uno strumento del `.NET` framwork di Microsoft. Esso mostra in fase di installazione una End-User License Agreement (EULA) in formato rtf (Rich text format).

2.3.4 Package per linux

Per facilitare l'installazione del compilatore Basalt su sistemi Linux, è stata predisposta la pacchettizzazione per snapcraft (package manager di canonical).

Snapcraft è uno strumento che consente di creare pacchetti snap, che sono pacchetti software sottoforma di container, che contengono tutte le dipendenze necessarie per l'esecuzione del software. Snapcraft è nato nell'ecosistema Ubuntu, ma è possibile utilizzarlo anche in altre distribuzioni Linux basate su Debian, e il supporto è attualmente in espansione anche per altre distribuzioni.

Il pacchetto non è attualmente sulle repository ufficiali di snapcraft, ma è possibile installarlo scaricandolo manualmente dalla repository github del progetto. Il nome per la futura pubblicazione di Basalt sui repository ufficiali di Snapcraft è già stato riservato.

2.4 Frontend: Tokenizzazione, Parsing, AST

In questo capitolo saranno approfondite le scelte implementative intraprese durante la realizzazione del frontend del compilatore.

2.4.1 Utilizzo di ANTLR nella repository *unina-Basalt*

Come già detto in precedenza, per la repository *unina-Basalt* è stato utilizzato ANTLR4 come frontend framework. Tale scelta implica l'utilizzo di un parser autogenerato.

In particolare, lo strumento da riga di comando `antlr4` è stato utilizzato per generare il codice sorgente della classe `BasaltParserVisitor`, il quale estende `AbstractParseTreeVisitor`.

La classe `BasaltParserVisitor` è stata pensata dagli sviluppatori di ANTLR per essere estesa da una implementazione concreta. Tale classe infatti espone dei metodi astratti che devono essere implementati per poter effettuare il parsing del codice sorgente.

Nel caso specifico di Basalt, la classe `BasaltParserVisitor` è stata estesa dalla classe `ConcreteBasaltParserVisitor`, la quale implementa i suddetti metodi astratti.

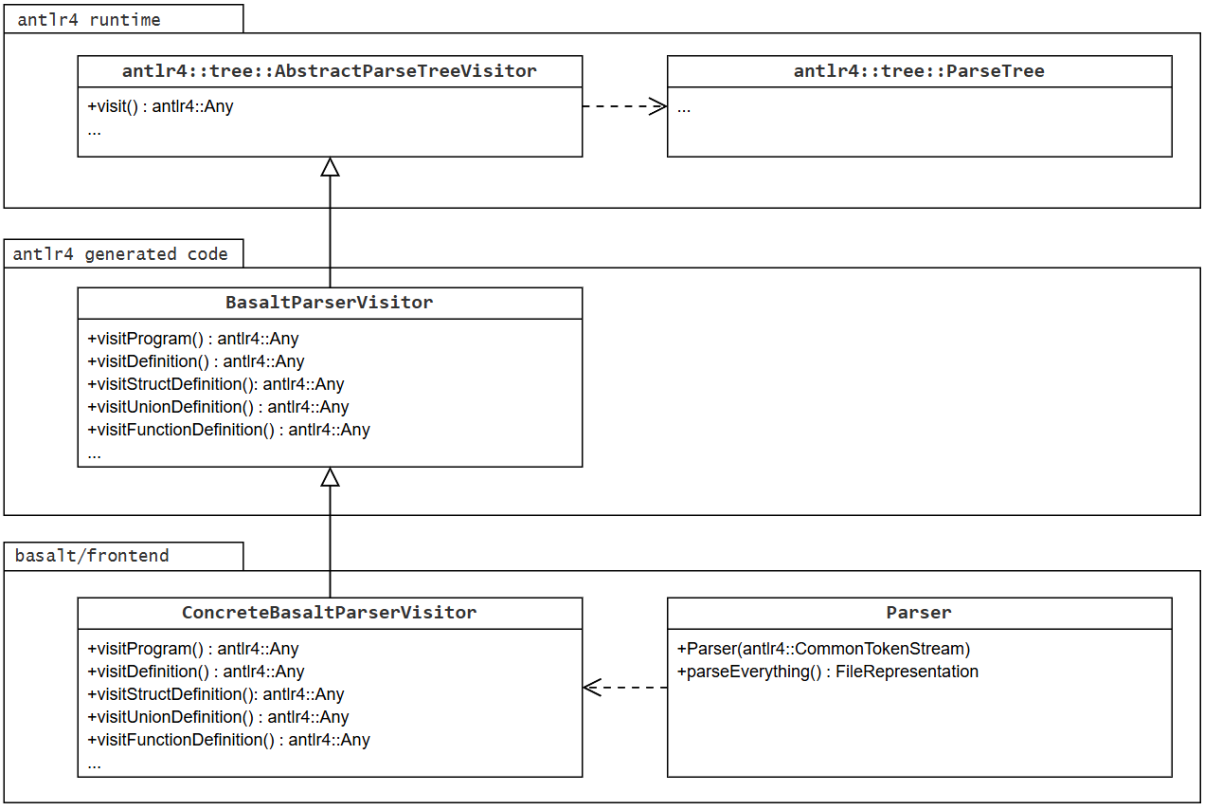


Figura 11: UML-Class-diagram del parser con ANTLR4

Il risultato di ogni chiamata ai vari metodi di `ConcreteBasaltParserVisitor` è un oggetto di tipo `antlr4::Any`, il quale è un tipo di dato fornito da ANTLR per modellare il risultato di una generica visita al parse tree.

Nel caso specifico, il risultato di ogni metodo di `ConcreteBasaltParserVisitor` è un'istanza di una classe definita nella codebase di Basalt, la quale rappresenta un'entità del linguaggio.

Utilizzando il metodo `parseProgram` il risultato sarà un'istanza di `FileRepresentation`, il quale codifica l'intero contenuto di un file sorgente.

Come input del costruttore della classe `Parser`, la quale è un wrapper su `ConcreteBasaltParserVisitor`, viene passato un oggetto di tipo `antlr4::CommonTokenStream`, il quale rappresenta il contenuto di un file sorgente.

Per ottenere un oggetto di tipo `antlr4::CommonTokenStream` è necessario utilizzare la classe `Tokenizer`, la quale è a sua volta un wrapper sulla classe autogenerata `BasaltLexer` (l'utilizzo dei wrapper serve ad esporre la stessa API della repository principale e consentire una migliore integrazione tra le due).

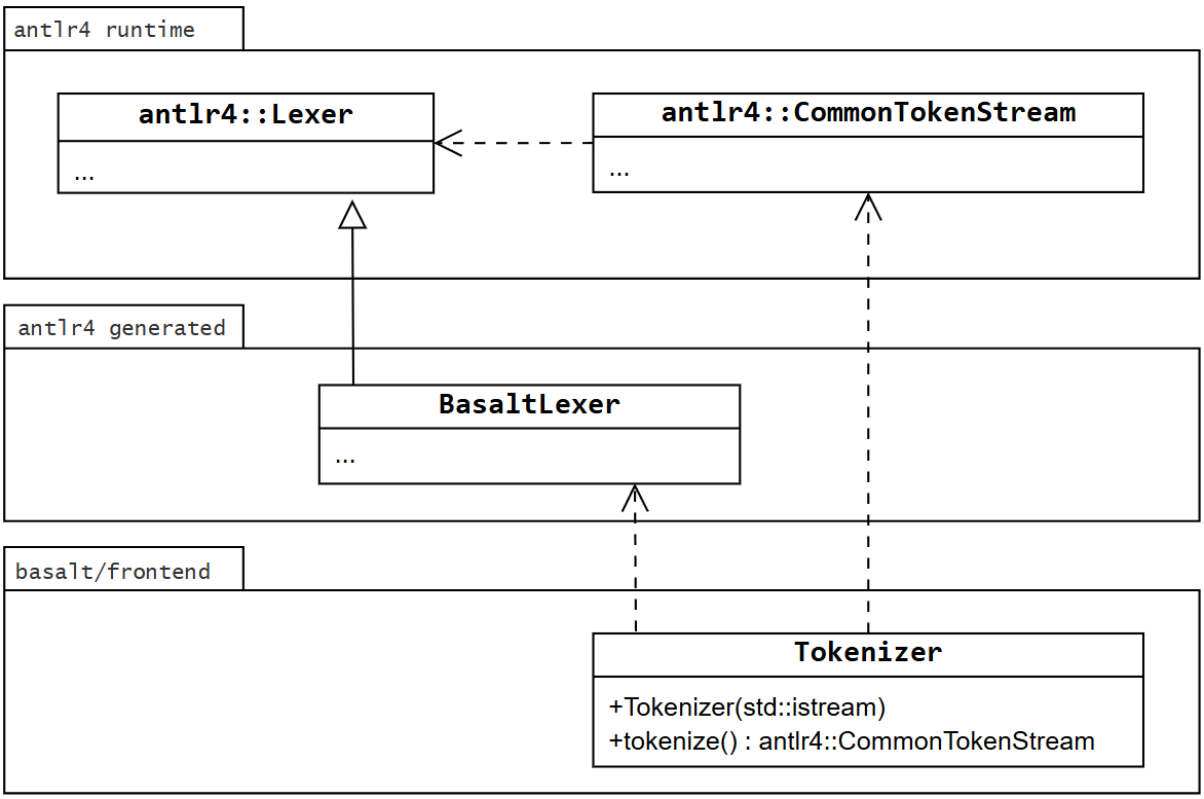


Figura 12: UML-Class-diagram del parser con ANTLR4

2.4.2 Tokenizzazione nella repository principale

Nella repository principale il tokenizer è stato realizzato su misura, senza l'uso di alcuna dipendenza esterna.

La classe `Tokenizer` espone due costruttori, uno per la costruzione a partire da un file ed uno per la costruzione a partire da una stringa , utilizzato durante il testing (in C++ è possibile trattare entrambi i casi in modo molto simile grazie alla classe `std::istream` che astrae sui dettagli implementativi sottostanti).
Il metodo `tokenize()` della classe tokenizer si occupa di effettuare la tokenizzazione dell'intero sorgente. È possibile schematizzare il funzionamento del metodo in pseudocodice come segue:

```
Tokenizer.tokenize():
    tokens = { }
    for line in source_code:
        char_index = 0
        while char_index < length(line):
            token, success = extract(line, &char_index)
            if (success):
                tokens.append(token)
    ensure_multiline_comments_closed()
    return tokens
```

Il metodo `tokenize()` si occupa di estrarre un singolo token per volta dal sorgente, esso è concretamente implementato come un dispatch sui vari metodi specifici per estrarre uno specifico tipo di token.

```
Tokenizer.extract(line, char_index):
    handle_multiline_comments(line, &char_index)
    handle_simple_comments(line, &char_index)
    first_char = line[char_index]
    tok = null
    switch(first_char):
        case '"' "'" '`': tok = extract_string(line, &char_index)
        case '0' ... '9': tok = extract_number(line, &char_index)
        case 'a' ... 'z': tok = extract_identifier(line, &char_index)
        case 'A' ... 'Z': tok = extract_typename(line, &char_index)
        case '+' ... '=': tok = extract_symbol(line, &char_index)
    return tok
```

I singoli metodi `extract_string`, `extract_number`, `extract_identifier`, `extract_typename`, `extract_symbol`, verificano se il testo della riga corrente, partendo dalla posizione `char_index`, coincida con quanto atteso. In caso di successo, essi restituiscono il token corrispondente alla porzione di testo che è stata riconosciuta e si occupano di avanzare l'indice fino al primo carattere non riconosciuto.

Il tokenizer mantiene traccia dei commenti multi-riga mediante uno stack. Esso viene popolato con i token relativi alle aperture dei commenti multi-riga (tali token sono salvati solo temporaneamente ai fini di error checking).

Ad ogni apertura di un commento multiriga, esso viene tokenizzato e salvato nello stack, mentre ad ogni chiusura di un commento multi-riga, si effettua un pop dallo stack.

```
Tokenizer.handle_multiline_comments(line, char_index):
    maybe_open_comment = line.substring(char_index, 2)
    if (maybe_open_comment == "/*"):
        comments_stack.push(make_token(line, char_index))
        char_index = char_index + 2
    while comments_stack.size() > 0 AND char_index < length(line):
        maybe_comment_seq = line.substring(char_index, 2)
        switch(maybe_comment_seq):
            case "/*":
                comments_stack.push(make_token(line, char_index))
                char_index = char_index + 2
            case "*/":
                comments_stack.pop()
                char_index = char_index + 2
            default:
                char_index = char_index + 1
```

Se al termine della tokenizzazione lo stack non dovesse essere vuoto, ciò significherebbe che almeno un commento multi-riga non è stato chiuso e i token che sono stati salvati nello stack potranno essere usati per fornire errori significativi in console.

Per quanto riguarda la gestione dei commenti su singola riga, essi sono gestiti in modo molto semplice. Qualora si incontri la sequenza di caratteri corrispondente all'apertura di un commento a singola riga, l'indice corrispondente al carattere corrente viene aggiornato in modo da far sembrare che la riga sia stata già tokenizzata fino al termine.

```
Tokenizer.handle_simple_comments(line, &char_index):
    maybe_open_comment = line.substring(char_index, 2)
    if (maybe_open_comment == "//"):
        char_index = length(line)
```

2.4.3 Parsing nella repository principale

Nella repository principale il parser è stato realizzato su misura, senza l'uso di alcuna dipendenza esterna. In particolare, il parser di Basalt è un parser *LL(*)* implementato mediante *recursive descent*.

Con *recurisve descent* si intende una tecnica implementativa di realizzazione di parser *LL(*)* che prevede l'utilizzo di funzioni (o metodi) i quali si chiamano a vicenda fra loro generando un grafo ricorsivo di chiamate.

Ad ogni chiamata viene passato, in termini di argomenti, tutto il necessario per tenere traccia del token corrente e dei token successivi durante il parsing. Ogni funzione si occupa di processare un certo numero di token a partire dal token corrente in avanti.

Ad esempio, per il parsing delle espressioni, si segue un approccio simile a quanto di seguito illustrato in pseudocodice, al netto di doverose ed opportune semplificazioni:

```
Parser.parse_expression(tok):
    Expression expr = parse_terminal_expression(&tok)
    if tok.current == BINARY_OPERATOR:
        expr = BinaryOperator {
            expr,
            tok.current(),
            parse_expression(tok.next())
        }
    return expr

Parser.parse_terminal_expression(tok):
    Expression expr = null
    switch token_stream_iterator.type:
        case IDENTIFIER:      expr = parse_identifier(&tok)
        case INT_LITERAL:     expr = parse_int_literal(&tok)
        case FLOAT_LITERAL:   expr = parse_int_literal(&tok)
        case BOOL_LITERAL:    expr = parse_bool_literal(&tok)
        case STRING_LITERAL:  expr = parse_string_literal(&tok)
        case ARRAY_LITERAL:   expr = parse_array_literal(&tok)
        case PREFIX_OPERATOR: expr = parse_prefix_operator(&tok)
        case OPN_PARENTHESYS: expr = parse_wrapped_expr(&tok)
    return expr

Parser.parse_prefix_operator(tok):
    UnaryOperator prefix_op = {
        tok.current,
        parse_expression(tok.next())
    }
    return prefix_op
```

2.4.4 Implementazione dell'AST

L'output della fase di parsing è un oggetto di tipo `FileRepresentation`. Esso è sostanzialmente la radice dell'AST relativo ad un singolo file sorgente, ed è composto da una lista di radici di sotto-alberi corrispondenti alle definizioni contenute nel file.

```
struct FileRepresentation {

    struct Metadata {
        std::string filename;
        std::string packagename;
        std::vector<std::string> imports;
    };

    Metadata file_metadata;
    std::vector<TypeDefinition> type_defs;
    std::vector<FunctionDefinition> func_defs;
};
```

La classe `TypeDefinition` rappresenta una definizione di tipo, ed è *super-tipo* di tutti nodi dell'AST che rappresentano definizioni di tipo.

Si tenga presente che non necessariamente essere super tipo in C++ significa essere esteso tramite ereditarietà. Nello specifico, per le definizioni di tipo, la classe `TypeDefinition` eredita dalla classe `std::variant`, la quale è sostanzialmente una union. Una `TypeDefinition` infatti è implementata come una union delle classi `UnionDefinition`, `StructDefinition` e `TypeAlias`.

Anche per le classe `TypeSignature`, `Expression` e `Statement` si è seguita la stessa logica, ovvero si è deciso di far si che esse fossero super-tipi di tutti i nodi dell'AST che rappresentano rispettivamente una firma di tipo, un'espressione o una istruzione, ma ciò non è stato ottenuto tramite ereditarietà. Per queste ultime si è scelto di procedere all'implementazione ispirandosi ad un design pattern proprio di C++, ovvero il patter *Type-Erasure*.

Tale design pattern è un wrapper su di una gerarchia classica ad oggetti basata su ereditarietà, ma che espone una API senza puntatori, e quindi molto più comoda da usare.

Si è però deviati dall'implementazione classica di tale pattern, in quanto si è scelto di conservare traccia del tipo sottostante per poter all'occorrenza verificare il tipo concreto dell'oggetto prima di procedere ad un cast.

Per tutto il resto della trattazione, si userà il termine *super-tipo* con l'accezione appena descritta, ovvero si dirà che se oggetti di tipo `T` sono assegnabili ad oggetti di tipo `U`, allora `T` è un super-tipo di `U`.

Alla luce di tale considerazione, si può procedere all'analisi delle classi che compongono l'AST illustrandone gli UML class diagram. Anche negli UML class diagram, la freccia che tipicamente indica l'ereditarietà sarà usata con l'accezione appena descritta.

Di seguito segue l'UML class diagram relativo alle classi che modellano le espressioni in forma di AST. Si tenga presente che anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

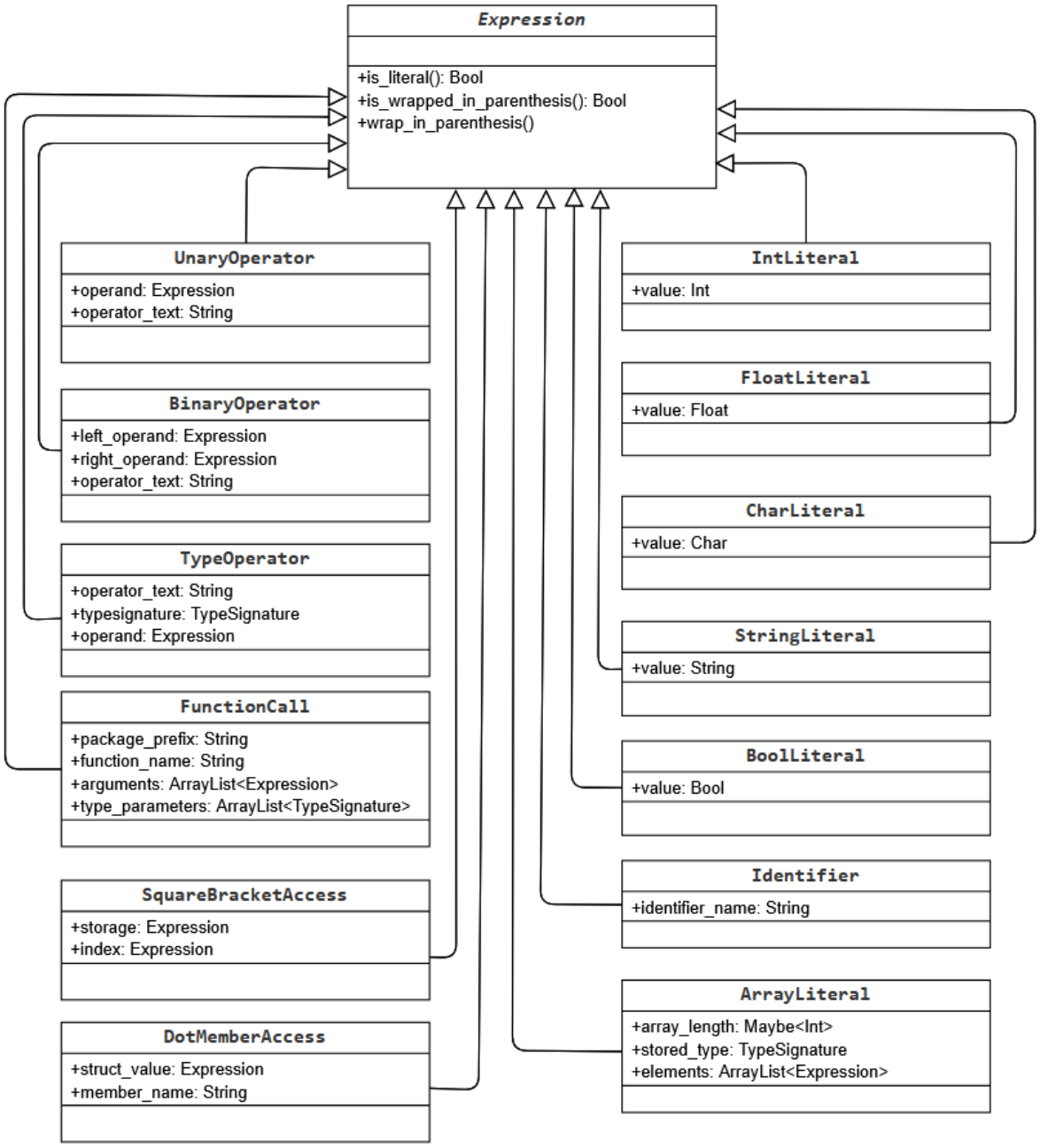


Figura 13: UML class diagram dei nodi dell'AST che rappresentano espressioni

Di seguito segue l'UML class diagram relativo alle classi che modellano gli statement in forma di AST. Così come già detto, anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

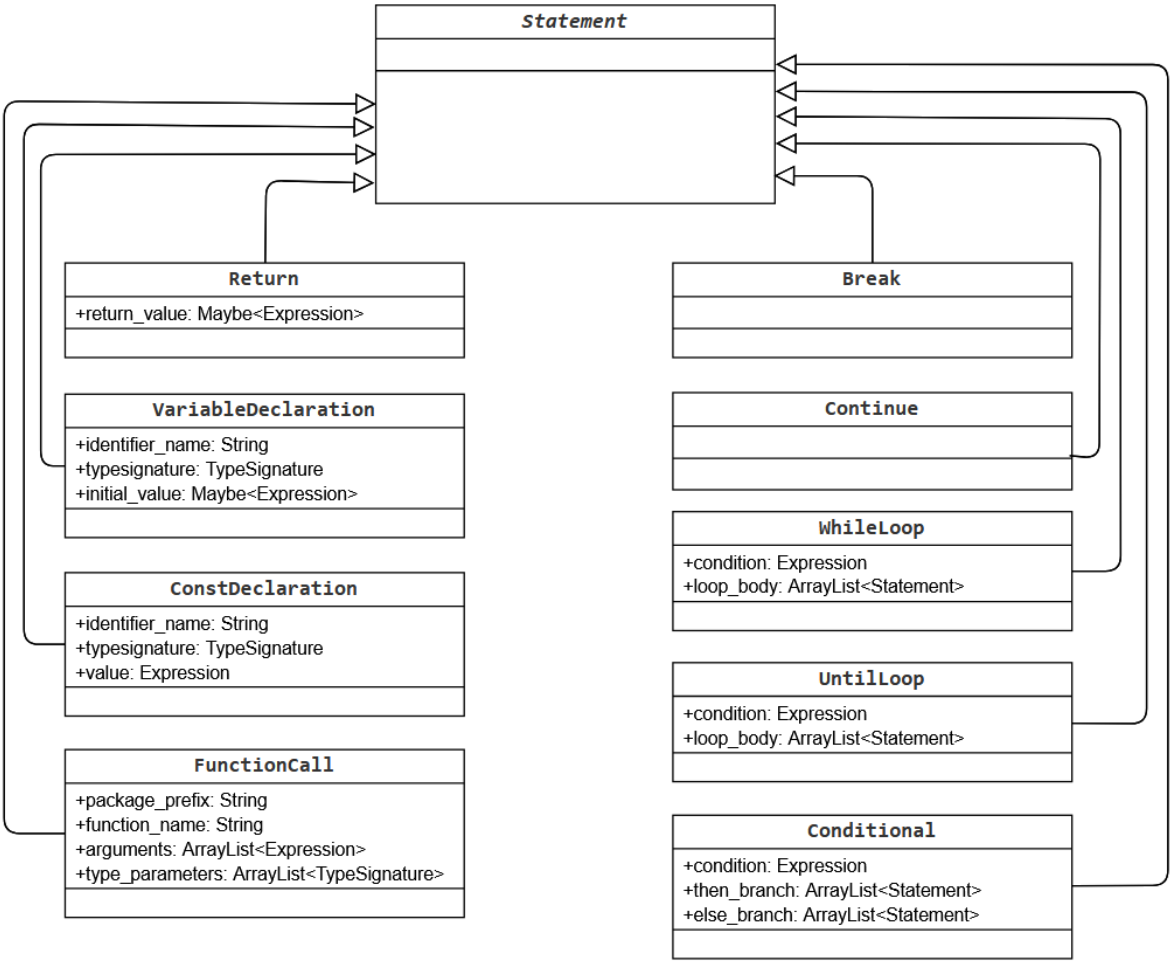


Figura 14: UML class diagram dei nodi dell'AST che rappresentano statement

Si può osservare come la classe `FunctionCall` sia un sottotipo di `Expression` e di `Statement`. Questo è dovuto al fatto che una chiamata di funzione può essere usata sia come espressione, se restituisce un valore, sia come statement, in caso contrario.

Si ricordi infatti che in C++ esiste l'ereditarietà multipla, e quindi è possibile che un una classe abbia più di un super-tipo anche nei casi in cui si usa l'eredita.

Di seguito segue l'UML class diagram relativo alle classi che modellano type-signatures e definizioni di tipo in forma di AST. Così come già detto, anche la repository basata su ANTLR utilizza la medesima rappresentazione, e si utilizza il visitor autogenerato da ANTLR per navigare la rappresentazione di ANTLR e tradurla opportunamente.

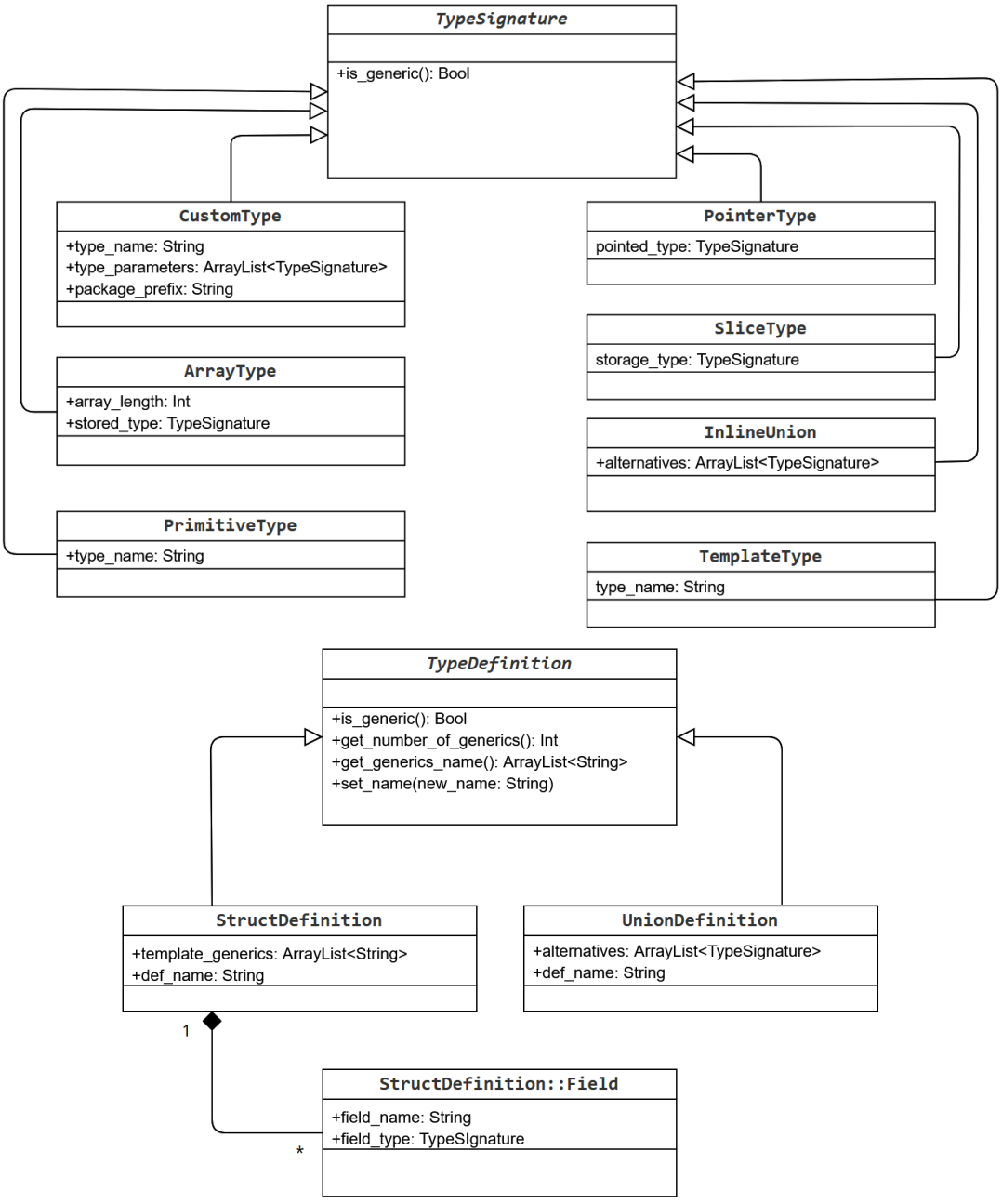


Figura 15: UML class diagram dei nodi dell'AST che rappresentano le type-signature e le definizioni di tipo

2.5 Logica interna: Symbol Tables, Typechecking, Reificazione

2.5.1 Merge degli output di parsing dei vari file sorgente

2.5.2 Costruzione della tabella dei tipi

2.5.3 Controllo di aciclicità delle dipendenze dirette fra tipi

2.5.4 Controllo di non-shadowing dei tipi

2.5.5 Tracciamento degli scope e delle definizioni locali

2.5.6 Typechecking

2.5.7 Algoritmo di Type-inference

2.5.8 Scoring degli overload

2.5.9 Costruzione della tabella delle funzioni

2.5.10 Generics: Sistema di reificazione

2.5.11 Gestione ad alto livello della CFA

2.5.12 Astrazione rispetto ad overload semplici ed overload CFA

2.6 Backend: Utilizzo di LLVM per generare IR

2.6.1 Traduzione dei tipi in LLVM-IR

2.6.2 Traduzione delle espressioni in LLVM-IR

2.6.3 Traduzione delle alterazioni del flusso di esecuzione in LLVM-IR

2.6.4 Traduzione delle funzioni in LLVM-IR

2.6.5 Cast impliciti

2.6.6 Operatore is ed operatore as

2.6.7 Implementazione della CFA