Il linguaggio Meucci

Versione 2.0.0

De Donato Paolo

Indice

Capitolo 1

Introduzione

1.1 Panoramica

Il linguaggio Meucci è incentrato principalmente sul concetto di **modulo** piuttosto alle funzioni del Lisp o agli oggetti del Java. Un modulo nel linguaggio Meucci è un **contenitore di funzioni e oggetti che svolgono una particolare funzione**. Il concetto di modulo è molto simile a quello di namespace in C++ o di package in Java, ma a differenza del Meucci questi linguaggi hanno posto maggior attenzione al concetto di oggetto (in particolare il Java), assegnando quindi ai moduli un ruolo marginale.

Ovviamente chi è del tutto estraneo al mondo della programmazione non avrà quasi sicuramente capito molto dalle righe precedenti. Per comprendere meglio i concetti di tipo (che in altri linguaggi equivarrebbe agli oggetti), funzione e modulo presenti nel linguaggio Meucci utilizzeremo il seguente paragone.

*Supponiamo che la nostra applicazione sia una fabbrica di liquori. Essa è composta da vari **moduli**, che equivarrebbero ai vari reparti della fabbrica. Prendiamo ora in esame il reparto di distillazione. I **tipi** potrebbero essere uno il "mosto" già fermentato ma che deve essere distillato, un altro il distillato, un altro gli aromi ... ovvero tutti gli oggetti concreti su cui è possibile effettuare operazioni di "lettura" (è possibile analizzare le sue caratteristiche) e/o "scrittura" (è possibile modificare le sue caratteristiche, magari combinandole con altri oggetti). Ovviamente non esiste un solo aroma, esiste ad esempio l'aroma della fragola, del limone, l'aroma della ciliega, etc. . Ciascuno di questi aromi corrisponderebbe all'interno del nostro programma ad una **variabile del tipo "aroma"**. Le **funzioni/operazioni** potrebbero essere i distillatori, le macchine di miscelazione, ... ovvero tutte le macchine che trasformano i prodotti grezzi (le variabili di un tipo prefissato passate alle funzioni), in prodotti elaborati (il valore di ritorno). In una grande fabbrica di liquori non esiste chiaramente solo il reparto di distillazione, esiste anche il reparto di fermentazione, il reparto di imballaggio e distribuzione, il reparto contabilità e il reparto che permette agli*

altri reparti di comunicare tra loro. Chiaramente, se l'azienda è molto piccola, è possibile utilizzare un solo reparto che effettui tutte queste operazioni, ma sarebbe tutto più incasinato e problematico, soprattutto all'aumentare della grandezza della fabbrica.

1.2 Compilazione

Per poter creare un eseguibile a partire dai sorgenti c'è bisogno di un particolare programma detto **compilatore**. L'attuale compilatore è il `mecc`, e i file contenenti i sorgenti devono avere come estensione `.x`. Supponiamo di voler compilare il file sorgente `Inizio.x` contenente le seguenti righe di codice:

```
modulo Inizio{
  int main(){
    println("Stringa stampata a video!");
    //Commento ignorato dal compilatore
  }
}
```

Se vogliamo compilare il programma su un sistema operativo Linux basterà navigare con la shell fino alla cartella contenente il nostro file e battere il seguente comando:

```
meucci@Meucci:/home/meucci/Code$mecc Inizio.x
```

otterremo in tal modo l'eseguibile `a.out`. Proviamo ad eseguirlo

```
meucci@Meucci:/home/meucci/Code$./a.out
Stringa stampata a video!
meucci@Meucci:/home/meucci/Code$
```

Analisi del programma

Analizziamo più nel dettaglio la struttura del programma. La prima riga `modulo Inizio` definisce un nuovo modulo di nome Inizio. Il corpo del modulo è contenuto all'interno di una coppia di parentesi graffe

```
{
...
}
```

In particolare il modulo contiene una funzione di nome `main` senza parametri di input e che ritorna un numero (un tipo particolare di numero in Meucii si esprime con `int`) che, tramite la funzione

```
println("Stringa stampata a video!");
```

stampa a video la stringa passata come parametro di input. La funzione `main` è particolare, in quanto il programma inizia l'esecuzione sempre dalla funzione con questo nome.

File generati

Dopo aver effettuato la compilazione, si noterà che è stato generato un nuovo file col nome `Inizio.in`. Questo file ha una funzione simile a quella dei file intestazione del C, ovvero permettere ad altri moduli di poter utilizzare le funzioni ed eventuali tipi del modulo `Inizio`.

Questo argomento verrà trattato più in dettaglio nel capitolo dei moduli.

Capitolo 2

Istruzioni fondamentali

2.1 Dichiarazione di variabili

Per comprendere più a fondo il funzionamento di un programma, dobbiamo conoscere alcune componenti del calcolatore (che può essere un normale computer desktop, un mainframe o un cellulare ...), in particolare la **memoria centrale** (conosciuta anche col nome di memoria RAM). Un qualunque programma infatti viene copiato all'interno della memoria centrale per poter essere eseguito dalla CPU e per poter memorizzare dati necessari alla corretta esecuzione. La memoria viene vista dalla CPU come se fosse una sequenza di byte, a ciascuno dei quali è associato un indirizzo univoco. Una **variabile** in Meucci (e in qualunque altro linguaggio di programmazione) non è nient'altro che un **indirizzo** di una **locazione di memoria** (uno o più bytes consecutivi) della memoria centrale, su cui possono essere eseguite le operazioni di **lettura dati** e di **scrittura dati**, che il programma utilizza per memorizzare temporaneamente un dato necessario all'elaborazione.

Per utilizzare una variabile è necessario che essa sia **dichiarata**, ovvero che il compilatore sia informato della sua esistenza affinché possa associargli un indirizzo in memoria. Nella dichiarazione di una variabile deve essere specificato il **nome** della variabile, che verrà utilizzato nel programma per accedere alla variabile, e il **tipo** di dati che la variabile può contenere. Il nome di una variabile può essere una qualunque sequenza di numeri o lettere, purché non contenga spazi e che il primo carattere sia una lettera. Inoltre il Meucci è **case sensitive** (ovvero distingue le lettere maiuscole da quelle minuscole). La seguente riga di codice mostra come si dichiara una variabile:

```
int a;
```

In questa riga di codice è stata dichiarata una variabile di tipo `int` e di nome `a`. In Meucci, come in ai devono creare delle funltri linguaggi, al termine di ogni istruzione bisogna porre il carattere `;` e, dato che la dichiarazione di una varia-

bile è essa stessa una istruzione, deve essere terminata da un punto e virgola. Analizziamo ora un esempio un po' più complesso:

```
int a;
int b;
String c;
a = 4;
b = a * 2 + 8;
```

In questo esempio sono state dichiarate tre variabili distinte: due di tipo `int` e una di tipo `String`. Per poter scrivere un dato in una variabile si usa il simbolo `=` con a sinistra il nome della variabile in cui scrivere il dato, e a destra il dato da scrivere nella variabile (nel nostro esempio abbiamo posto nella variabile `a` il valore intero `4`). A destra dell'operatore di assegnazione è possibile inserire espressioni anche più complesse, nell'ultima riga infatti il programma effettua le seguenti operazioni:

1. Legge il valore contenuto nella variabile `a`
2. Lo moltiplica per `2`
3. Somma al risultato il numero `8`
4. Scrive il risultato nella variabile `b`

Alla fine dell'esecuzione delle istruzioni precedenti la variabile `a` conterrà il valore `4`, mentre la variabile `b` il valore `16`. Quando si dichiara una nuova variabile essa si trova in uno stato **indefinito** (ovvero non è possibile determinare il suo contenuto), per questo prima di ogni utilizzo essa si deve **inizializzare**, ovvero bisogna scriverci dentro un valore noto. In Meucci è possibile dichiarare e inizializzare una variabile in una sola istruzione, in tal modo l'esempio precedente diviene:

```
int a = 4;
int b = a * 2 + 8;
String c;
```

2.1.1 Dichiarazioni auto

In una dichiarazione bisogna sempre specificare il tipo di una variabile. Spesso, soprattutto se si utilizzano parametri template, il nome di un tipo risulta essere troppo lungo. In Meucci è possibile utilizzare in una dichiarazione con assegnazione al posto del nome del tipo l'identificativo **auto**, che dice al compilatore di determinare il tipo della variabile direttamente dal tipo di ritorno dell'espressione di assegnamento. Dunque l'esempio seguente

```
auto x = 32;
```

dichiara una variabile di tipo `int` (vedere il prossimo paragrafo) e gli assegna il valore intero 32. Ovviamente la dichiarazione tramite l'identificativo auto richiede un assegnazione, altrimenti il compilatore genererà un errore.

2.1.2 Visibilità delle variabili

Un **blocco di istruzioni** è una seguenza di istruzioni racchiuse tra parentesi graffe. Il corretto utilizzo dei blocchi aumenta la leggibilità del codice e permette anche di ridurre la memoria inutilizzata tramite il concetto di **visibilità** (o **scope**) di una variabile.

In parole povere, se una variabile viene dichiarata all'interno di un blocco di istruzioni allora sarà accessibile solo all'interno di tale blocco. In tal modo il compilatore potrà liberare la memoria riservata alle variabili dichiarate all'interno del blocco una volta usciti dallo stesso, riutilizzando la memoria per le altre variabili.

2.2 Tipi primitivi e costanti

Nell'esempio precedente abbiamo utilizzato il tipo `int` dando per scontato che fosse il tipo di una variabile numerica. In Meucci esistono due classi di tipi in base a come vengono memorizzati i dati all'interno della memoria: i tipi **primitivi** sono i più semplici poiché l'indirizzo associato alla variabile punta direttamente ai dati su cui operare; i tipi **reference** invece contengono un **puntatore** ai dati. Più avanti tratteremo dei tipi reference, mentre in questa sezione ci concentreremo sui tipi primitivi.

I tipi primitivi sono determinati direttamente dal compilatore, e sono i seguenti:

- `byte` un numero intero con segno memorizzato in 1 byte (compreso tra -128 e 127 inclusi)
- `short` un numero intero con segno memorizzato in 2 bytes (compreso tra -32.768 e 32.767)
- `int` un numero intero con segno memorizzato in 4 bytes (compreso tra -2.147.483.648 e 2.147.483.647)
- `long` un numero intero con segno memorizzato in 8 bytes (compreso tra -9.223.372.036.854.775.808 e 9.223.372.036.854.775.807), disponibile solo per architetture a 64 -bit
- `ubyte` un numero intero senza segno memorizzato in 1 byte (compreso tra 0 e 255)
- `ushort` un numero intero senza segno memorizzato in 2 bytes (compreso tra 0 e 65.535)

- `uint` un numero intero senza segno memorizzato in 4 bytes (compreso tra 0 e 4.294.967.295)
- `ulong` un numero intero senza segno memorizzato in 8 bytes (compreso tra 0 e 18.446.744.073.709.551.615), disponibile solo per architetture a 64 -bit
- `char` un carattere contenuto in 1 byte (come per la codifica ASCII)
- `real` un numero a virgola mobile con precisione doppia (52 bit per mantissa, 11 bit per l'esponente)
- `pt` un puntatore generico (come il `void *` del linguaggio C), la dimensione dipende dall'implementazione
- `boolean` un valore booleano (vero o falso) contenuto in 1 byte

Le **costanti** sono valori stabiliti a tempo di compilazione, `2, 4, 8` sono un esempio di costanti. Come per le variabili, anche le costanti hanno un tipo, determinabile secondo le seguenti regole:

- Una `u` o `U` davanti ad una costante numerica intera esprime che il tipo non ha segno, altrimenti ha segno;
- Se una costante numerica termina con `b` o `B` allora è `byte` o `ubyte`, se termina con `s` o `S` allora è `short` o `ushort`, se termina con `l` o `L` allora è `long` o `ulong`, se non è presente è un `int` o `uint`
- Un carattere ASCII deve essere compreso tra due apici singoli (')
- Una costante a virgola mobile deve contenere un punto, seguito da almeno una cifra (anche 0)

Ecco un esempio di costanti:

```
43
u12S
76b
45.0
U11
908753129L
'h'
123.45632
```

che sono di tipo rispettivamente: `int`, `ushort`, `byte`, `real`, `uint`, `long`, `char`, `real`. L'unica costante per `pt` è `null`, che rappresenta sempre un indirizzo invalido (esistono altri modi per assegnare a `pt` un indirizzo valido).

2.3 Conversioni di tipo

Il linguaggio di programmazione Meucci presenta un meccanismo molto forte di protezione dei tipi: ad ogni variabile di un determinato tipo è possibile assegnarli solo costanto od espressioni dello stesso tipo: le seguenti istruzioni infatti genereranno errori di compilazione:

```
int a = u4;
```

```
int a = 4;
long b = a;
```

l'errore è generato nella seconda riga

```
int a = 4 + 12b;
```

L'operazione `+` , come altre operazioni, richiede che i due termini siano dello stesso tipo.

Ovviamente esistono casi in cui bisogna cambiare temporaneamente il tipo di un'espressione per poter essere utilizzato in opportuni contesti (basta pensare alle funzioni). In tal caso è possibile utilizzare il **casting** del tipo: basta far precedere all'espressione che si vuole convertire con il nome del nuovo tipo racchiuso tra parentesi. Il secondo esempio può allora essere corretto nel seguente modo:

```
int a = 4;
long b = (long) a;
```

Se l'operatore di cast è utilizzato tra tipi numerici entrambi con o senza segno, e inoltre il nuovo tipo ha una dimensione maggiore o uguale al tipo precedente allora vi è la garanzia che il valore numerico e l'eventuale segno siano mantenuti. Se invece la conversione avviene tra un tipo numerico con segno e uno senza segno allora il numero viene esteso tenendo conto del nuovo tipo (se è senza segno viene esteso tramite zeri, altrimenti secondo la specifica del processore per i numeri con segno). Se il nuovo tipo numerico ha dimensione minore del vecchio tipo numerico, allora vengono troncate le cifre più significative del numero. Infine, nelle conversioni tra tipi non numerici la situazione è più complessa. In particolare la conversione da `real` ad un tipo intero **NON determina la parte intera del numero**, se si vuole calcolare la parte intera di un numero a virgola mobile si dovranno utilizzare opportuni algoritmi. Lo stesso discorso si applica per conversioni da tipi numerici a `real`.

2.4 Operazioni elementari

Gli algoritmi per poter operare sui dati vengono eseguiti da due strutture: le funzioni e le operazioni. Le operazioni sono definite tramite un nome, che può essere o composto esclusivamente da simboli

`| $ % & + - * / \ ? ^ < > = | ~ @`

oppure da una stringa preceduta da : (Es. `:new`). Un operazione può operare o su un solo parametro (il simbolo va posto prima del parametro) oppure due (il simbolo viene posto tra i due parametri). Di seguito elenchiamo le principali operazioni in Meucci incluse direttamente nel compilatore:

Operazioni con 2 parametri:

- `+ - * / :mod` Effettuano rispettivamente addizione, sottrazione, moltiplicazione, divisione e calcolo del resto tra tipi numerici interi dello **stesso tipo**
- `& | :xor` Effettuano rispettivamente and, or ed or esclusivo bit a bit di tipi numerici (sempre dello stesso tipo)
- `== < > <= >= !=` Confronto di numeri interi dello stesso tipo
- `&& || ->` Acettano due espressioni logiche (che ritornano `boolean`). `&&` ritorna `true` se e solo se entrambe le espressioni logiche ritornano `true`; `||` ritorna `true` se e solo se almeno una espressione ritorna `true`; `->` ritorna `false` se e solo se la prima espressione (a sinistra del simbolo) ritorna `true` e la seconda ritorna `false`
- `<< >>` Effettua uno shift a sinistra e a destra di tanti bit quanto è il valore ritornato dalla seconda espressione. La prima espressione deve ritornare un tipo numerico, mentre la seconda deve ritornare necessariamente `ubyte`
- `+ - * /` Queste espressioni, oltre che con tipi numerici interi, possono essere utilizzate anche se le espressioni ritornano `real`

Operazioni con 1 parametro:

- `++ -- - ~` Applicato ad espressioni numeriche, rispettivamente incrementano e decrementano di un unità, effettuano un complemento a 2 (opposto di un numero) e il complemento a 1. A differenza del C/C++, le prime due operazioni non modificano l'espressione su cui sono utilizzate
- `!` Applicato ad espressioni logiche, ritorna il negato
- `:sqrt` Applicato a espressioni `real`, ritorna la radice quadrata

Esempi:

```
int a = 4 - 8;
int b = a << u1b;
boolean x = (++ a > b) -> (12 != 12);
```

A differenza di altri linguaggi di programmazione, in Meucci le regole di associatività sono solo le seguenti:

1. Le operazioni con un solo parametro hanno sempre la precedenza sulle operazioni a 2 parametri
2. Se più operazioni ad un parametro sono applicate alla stessa espressione allora vengono eseguite da destra a sinistra
3. Le operazioni a due parametri vengono eseguite sempre da sinistra a destra

Quest'ultimo punto può generare molta confusione ai programmatori, soprattutto ai più esperti. Per esempio l'espressione

```
3 + 4 * 6
```

ritorna 42 invece di 27 come per il C. Per ovviare a questo inconveniente si possono utilizzare le parentesi tonde. Infatti

```
3 + (4 * 6)
```

ritornerà 27. Più avanti verrà chiarito il motivo che ha portato all'adattamento di questa politica.

Elisione del primo parametro

Supponiamo di avere queste righe di codice:

```
int a = 4;
a = a + 2;
```

Dato che, nella seconda riga, l'espressione a sinistra dell'operazione è la stessa presente a sinistra dell'operazione di assegnamento, è possibile rimuovere la prima espressione, in modo da ottenere

```
int a = 4;
a =+ 2;
```

che produrrà lo stesso risultato. Si nota che il simbolo dell'operazione deve seguire il simbolo di assegnazione direttamente e senza spazi, altrimenti verrà interpretato come un'operazione ad un parametro.

2.5 Costrutti di controllo ed iterazione

2.5.1 Costrutto if ed if-else

Fino ad ora abbiamo analizzato le principali istruzioni presenti in Meucci. Queste istruzioni verranno eseguite sequenzialmente all'interno del programma, ma molto spesso si vuole che alcune istruzioni vengano eseguite solo in particolari condizioni. Come in tutti gli altri linguaggi di programmazione, il linguaggio

Meucci mette a disposizione il costrutto `if` che analizza una particolare espressione logica ed esegue le istruzioni specificate solamente se l'espressione logica ritorna `true`.

Ecco un esempio del'utilizzo del costrutto `if`:

```
int x;
...
if(x > 0){
 x =- 1;
}
```

il programma esegue l'istruzione `x =- 1` se e solo se l'espressione logica `x > 0` è verificata.

Analogamente è presente il costrutto `if-else`:

```
int x;
...
if(x > 0){
 x =- 1;
}
else{
 x =+ 1;
}
```

in questo esempio se `x > 0` allora esegue `x =- 1`, altrimenti esegue `x =+ 1`.

2.5.2 Costrutti while e for

Il costrutto **while** richiama più volte un gruppo di istruzioni finché l'espressione logica specificata non ritorna `false`.

```
int x;
...
while(x > 0){
 x =- 1;
}
```

In questo esempio il programma richiama l'istruzione `x =- 1` fino a quando l'espressione `x > 0` non ritorna `false`, in particolare se `x` è positivo verrà decrementato fino a diventare 0, e quindi esce dal ciclo. Da notare che `while` controlla l'espressione prima di eseguire il blocco di istruzioni associato, quindi se `x` contiene un numero negativo o 0 viene lasciato inalterato.

Analogo al costrutto while c'è anche il costrutto **for**, anch'esso è un costrutto di iterazione, ma a differenza del `while` oltre al blocco di istruzioni da iterare accetta altre tre istruzioni come parametri: la prima deve essere un'espressione, una dichiarazione o un assegnamento che viene eseguita prima di entrare nel blocco, la seconda un espressione logica mentre la terza una espressione

generica o un assegnamento che viene eseguita prima del controllo (detta istruzione di **step**). Questi parametri devono essere sempre separati con un ; . Per comprenderne meglio il funzionamento, osserviamo che il seguente costrutto:

```
for(int x = 0; x < 10; x =+ 1){
 println(x);
}
```

è del tutto equivalente a

```
int x = 0;
while(x < 10){
 println(x);
 x =+ 1;
}
```

Le variabili dichiarate nell'intestazione del costrutto for, così come quelle definite all'interno di strutture if, while e for, hanno la visibilità limitata alla struttura stessa, e quindi non possono essere utilizzate all'esterno.

2.5.3 break e continue

All'interno dei costrutti di iterazione sono disponibili due nuove istruzioni:

- `break` Esce immediatamente dal ciclo
- `continue` In un costrutto while salta direttamente alla valutazione dell'espressione logica, mentre in un costrutto for salta all'istruzione di step.

Capitolo 3

Moduli

3.1 Creazione di un modulo

Nell'introduzione abbiamo definito i moduli in Meucci. Ogni funzione, operazione e tipo deve essere contenuto in qualche modulo, e in ogni file ci può essere uno ed un solo modulo. Per creare un modulo si utilizza la parola chiave **modulo**, seguita dal nome del modulo e dal corpo (contenente le funzioni ed i tipi) racchiuso da una coppia di parentesi graffe

```
modulo Mode{
...
...
}
```

3.1.1 Importare altri moduli

Una caratteristica dei moduli in Meucci è la capacità di importare le funzionalità di altri moduli. Per importare moduli è necessario possedere, oltre alla libreria contenente i codici, anche il relativo file di importazione (terminante con .in) generato durante la compilazione, che si deve trovare in una posizione nota al compilatore.

Per segnalare al modulo di importare un altro modulo si utilizza la parola chiave **depends** seguita dai moduli da importare, seguiti da uno spazio.

```
modulo Mode depends Mod1 Mod2 Mod3{
...
```

3.2 Variabili esterne ed inizializzatore

Spesso un modulo per svolgere le sue funzionalità deve immagazzinare dati che non devono essere distrutti al termine delle funzioni e non possono essere pas-

sati alla funzione chiamante. Ciò può essere ottenuto definendo delle **variabili esterne**, ovvero variabili immagazzinate in una parte della memoria a cui si può accedere solo tramite funzionalità interne al modulo e sopravvivono per tutta la durata del programma.

Una variabile esterna si dichiara come una normale variabile all'interno del corpo del modulo

```
modulo Mode{
 int vesterna1;
 ushort vest2;
 ...
 ...
}
```

Per inizializzare queste variabili si può utilizzare un blocco di istruzioni **static**, che viene eseguito una sola volta che il modulo è caricato in memoria:

```
modulo Mode{
 int vesterna1;
 ushort vest2;
 static{
  vesterna1 = 0;
  vest2 = u3s;
 }
 ...
 ...
}
```

Capitolo 4

Funzioni ed operazioni

4.1 Definire funzioni ed operazioni

Spesso è utile racchiudere una o più istruzioni all'interno di una **funzione** o di un'**operazione** affinché possano essere utilizzate in altri punti del programma senza il bisogno di ricopiare ogni volta tutte le istruzioni, con una significativa riduzione del codice.

Una funzione può avere dei **parametri** che possono influenzare le operazioni svolte dalla funzione e che vengono trattati all'interno della funzione come delle variabili. Inoltre una funzione può avere o no un **valore di ritorno**, ovvero il risultato che deve essere passato al programma chiamante una volta terminata l'elaborazione della funzione.

Per definire una funzione bisogna prima scrivere il **tipo** del valore di ritorno (o **void** se la funzione non ritorna niente), poi il nome della funzione e infine gli eventuali parametri della funzione, separati da una virgola e racchiusa tra parentesi. Quindi scrivere le istruzioni all'interno di parentesi graffe. Per uscire dalla funzione e ritornare il valore di ritorno di un espressione si utilizza l'istruzione `return` seguita dall'espressione. In funzioni che non ritornano niente si può utilizzare l'istruzione return senza alcuna espressione.

Ecco un esempio in cui vengono dichiarate delle funzioni:

```
ubute A(){
 return u1b;
}

int valoreassoluto(int a){
 if(a >= 0)
  return a;
 else
  return - a;
}
```

```
void niente(ushort a, ushort b){
 if(a < u200S)
  return;
 a =+ b;
}
```

In Meucci è possibile definire anche le operazioni in modo analogo alle funzioni, ricordando che possono avere o 1 o 2 parametri solamente.

```
long +-@ (long a, long b){
 ...
}
```

Il passaggio di parametri a funzioni ed operazioni è sempre per valore, e mai per riferimento.

4.2 Overloading di funzioni ed operazioni

Come in C++, in Meucci è supportato l'overloading di funzioni. Per comprendere questo concetto prendiamo in esempio il linguaggio C. In C non è possibile definire due funzioni con lo stesso nome, in quanto il compilatore distingueva le funzioni solamente in base al nome. In C++ e in Meucci il compilatore distingue le funzioni non solo in base al nome, ma anche in base ai parametri di input della funzione, permettendo quindi al programmatore di definire funzioni simili con lo stesso nome e con parametri diversi senza che il compilatore generi errore. Ogni volta inoltre che si chiama la funzione il compilatore sceglie in base ai parametri passati quale funzione richiamare.

Il parametro di ritorno non viene considerato dal compilatore per distinguere le funzioni, quindi le seguenti funzioni:

```
...
int funz(ubyte a)
...
uint funz(ubyte v)
...
```

faranno generare al compilatore un errore.

4.3 Ricorsione

Le funzioni in Meucci supportano la ricorsione, ovvero una funzione può richiamare se stessa. Gli algoritmi ricorsivi sono molto utilizzati in quanto permettono di implementare algoritmi semplici e facilmente leggibili, nonostante a volte siano più lenti dei loro corrispettivi che utilizzano strutture iterative al posto della ricorsione.

L'esempio utilizzato per mostrare la ricorsione è l'algoritmo ricorsivo per il calcolo del massimo comune divisore di due numeri interi senza segno

```
uint MCD(uint a, uint b){
 if(b == u0)
  return a;
 else
  return MCD(b, a :mod b);
}
```

4.4 Gestione errori

Nella versione 2.0.0 del compilatore è possibile gestire gli errori (chiamati eccezioni in C++ e Java) **solamente per funzioni ed operazioni**. Spesso infatti durante l'esecuzione di un programma possono nascere degli errori che possono essere previsti ma non risolti durante la compilazione (basti pensare agli errori dovuti all'errato input dell'utente). Per poter gestire queste condizioni eccezionali è possibile rendere una qualunque funzione o operazione capace di **generare errori**. Molti programmatori alle prime armi sottovalutano l'utilità della gestione degli errori, ma col crescere delle dimensioni dei programmi cresce conseguentemente la necesstià della gestione degli errori.

4.4.1 Dichiarazione e generazione di errori

Per rendere una funzione (od operazione) capace di generare errori è necessario specificare i possibili errori che essa genera. A differenza di altri linguaggi come il C++ o il Java in cui le eccezioni sono particolari classi, in Meucci gli errori non sono oggetti a sé stanti, e quindi il programmatore non deve creare nuovi oggetti per ogni errore generato, con un conseguente guadagno in prestazioni.

Per dichiarare gli errori che una funzione può generare bisogna inserire tra la parentesi tonda chiusa e la parentesi graffa aperta la parola chiave **errors** seguita dai nomi degli errori separati da uno spazio. I nomi degli errori seguono le stesse regole dei nomi delle variabili.

```
void funzione(int par1, ulong par2)errors err1 err2 err3{
 ...
```

Per generare un errore si utilizza l'istruzione **throw** seguita dal nome dell'errore da generare e da un punto e virgola:

```
throw err2;
```

È possibile generare un errore non dichiarato dalla funzione, ma ciò causerà l'**immediata terminazione** del programma, in quanto non puÃš essere gestito.

4.4.2 Gestione errori

Se vogliamo utilizzare una funzione che potrebbe generare errori possiamo seguire due strade:

1. Dichiarare gli stessi errori nella funzione utilizzatrice facendo propagare l'errore;
2. Utilizzare un blocco try-catch per gestire l'errore.

Un blocco **try-catch** è uno speciale blocco che cattura e gestisce gli errori generati. Esso è composto da:

- Un blocco **try**: è semplicemente un blocco di istruzioni. Quando una delle istruzioni contenute genera un errore salta l'esecuzione al blocco catch che gestisce il relativo errore;

```
try{
 istruzione1
 istruzione2
 ...
}
```

- Un blocco **catch**: ci possono essere più blocchi catch, ciascuno che gestisce un errore diverso. Le istruzioni contenute in esso verranno eseguite solamente se è stato generato il relativo errore all'interno del blocco try;

```
catch(errName){
 istruzione1
 istruzione2
 ...
}
```

- Opzionalmente un blocco **default**: tale blocco viene eseguito solamente se è stata generato un errore che non può essere gestito da nessun blocco catch.

```
default{
 istruzione1
 istruzione2
 ...
}
```

Esempio: Nel seguente esempio è stato utilizzato un blocco try-catch per controllare l'esecuzione dell'istruzione `int var = funzione();`:

```
try{
 int var = funzione();
}
catch(Err1){
 //Se viene generato Err1 l'esecuzione passa qui:
 println("Err1");
}
catch(Err2){
 //Se viene generato Err2 l'esecuzione passa qui:
 println("Err2");
}
default{
 //Se viene generato un errore che non e'
 //Err1 ne' Err2 l'esecuzione passa qui:
 println("ErrGenerico");
}
```

Capitolo 5

Tipi reference

5.1 Creazione di tipi reference

Precedentemente abbiamo visto quali sono i tipi primitivi in Meucci, in particolare abbiamo visto che non è possibile aggiungere nuovi tipi primitivi senza aggiornare il compilatore. I tipi reference invece possono essere creati direttamente dal programmatore e memorizzati in una libreria per poter essere utilizzati in seguito.

Abbiamo già detto che quando quando si dichiara una variabile di tipo reference viene allocata memoria per il **puntatore** che punta ai dati veri e propri del tipo. Un tipo reference è composto da una o più variabili chiamate **campi** del tipo, che insieme immagazzinano tutte le informazioni associate al tipo reference.

Per comprendere megio questi concetti utilizzeremo un esempio classico. Supponiamo di voler creare il tipo Punto, ovvero un punto su di un piano cartesiano reale, che è identificato univocamente dalla coordinata delle ascisse (x) e quella delle ordinate (y). Il nostro tipo Punto verrà creato così:

```
type Punto{
 real x;
 real y;
}
```

Per creare un nuovo tipo reference bisogna utilizzare l'identificatore **type**, seguito dal nome del tipo, e quindi da ciascun campo terminato con un ; .

A ciascun campo possono essere associati dei modificatori, divisi in due categorie:

- I modificatori di **memorizzazione** esprimono come è memorizzato il campo all'interno del tipo. Gli attuali modificatori di memorizzazione sono `explicit`, `ghost` e `override`

- I modificatori di **accesso** esprimono come deve avvenire l'accesso al campo. Gli attuali modificatori di accesso sono `read` e `shadow`

Ogni campo può avere al più un modificatore di memorizzazione e al più un modificatore di accesso, e il modificatore di memorizzazione deve sempre precedere l'eventuale modificatore di accesso.

Più avanti analizzeremo nel dettaglio le funzioni di ciascun modificatore.

5.2 Variabili reference

Come per i tipi primitivi, è possibile dichiarare variabili di tipo reference come per i tipi primitivi:

```
type Frazione{
 int num;
 int den;
}
...
...
Frazione var;
```

Per accedere ai campi di una variabile reference si utilizza il simbolo punto (.) seguito dal nome del campo a cui accedere:

```
int i = var.num;
var.den = - 234;
```

5.2.1 Allocazione di oggetti

Quando dichiariamo una variabile reference in realtà stiamo creando una variabile contenente un puntatore che dovrebbe puntare al relativo oggetto. Dato che non abbiamo ancora creato un oggetto idoneo il puntatore punterà ad una locazione di memoria casuale e non prevedibile.

Per evitare la sovrascrittura di dati fondamentali al programma (in casi estremi si potrebbe arrivare anche al crash del sistema) è opportuno inizializzare il puntatore per farlo puntare ad un indirizzo sicuramente non valido. Ciò può essere ottenuto tramite l'espressione costante **null**, che può essere utilizzato anche per inizializzare una variabile `pt`.

```
Frazione var = null;
pt pointer = null;
```

Ovviamente una variabile inizializzata a `null` è nella maggior parte dei casi inutile. Meucci però possiede dei "pseudo-operatori" che allocano l'oggetto in memoria e ritornano un puntatore all'oggetto, che può essere assegnato alla relativa variabile:

1. **:stack** Alloca l'oggetto direttamente nello stack. L'allocazione viene effettuata molto velocemente ma viene distrutto una volta usciti dal blocco in cui è stato chiamato

2. **:new** Alloca l'oggetto nello heap. L'allocazione è più lenta rispetto a :stack, ma l'oggetto sopravvive al blocco in cui è stato creato e quindi può essere ritornato da una funzione. Per essere distrutto bisogna utilizzare la pseudo-operazione **:destroy**

3. **:static** L'allocazione è immediata ma al contempo sopravvive al blocco in cui è creato. L'oggetto viene creato una ed una sola volta, anche se l'istruzione si trova all'interno di un ciclo o di una funzione chiamata più volte. Non deve essere distrutto, se si prova a distruggere un oggetto allocato con :static si andrà incontro quasi sicuramente ad un errore logico. Utilizzato soprattutto negli inizializzatori dei moduli.

Ecco alcuni esempi di utilizzo:

```
Frazione f = :stack Frazione();
Ref g = :new Ref();
Frazione h = :static Frazione();
...
:destroy g;
```

5.2.2 Costruttori e distruttori

Se si prova ad eseguire il codice precedente il compilatore genererà un errore. Difatti :new, :stack e :static allocano lo spazio necessario all'oggetto ma non inizializzano i campi, che sono lasciati in uno stato indefinito. Per inizializzare un oggetto bisogna utilizzare una particolare funzione definita all'interno del tipo, detto **costruttore**, che si occuperà di inizializzare correttamente i campi. Per accedere ai dati dell'oggetto si deve utilizzare la variabile **this** dello stesso tipo dell'oggetto da inizializzare. Un costruttore può avere dei parametri di input come per le funzioni, che verranno passati durante l'allocazione della memoria, e in un tipo ci possono essere più costruttori distinti, purchè abbiano diversi parametri.

Oltre al costruttore in un oggetto deve essere definito anche un **distruttore**, che verrà chiamato quando verrà liberata la memoria associata all'oggetto. A differenza del costruttore vi potrà essere un solo distruttore per ciascun oggetto e non può avere parametri, anche se si puac o utilizzare this per accedere all'oggetto.

Un costruttore deve avere come nome **init**, mentre un distruttore **end**, in entrambi il tipo di ritorno **non** deve essere specificato (ma può essere considerato `void`) e devono terminare con un punto e virgola.

```
type Frazione{
 int num;
 int den;
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
...
...
Frazione f = :new Frazione(1, 2);
Frazione g = :stack Frazione(3);
```

5.3 Funzioni di accesso e modificatori explicit e ghost

Prendiamo sempre in esempio il tipo Frazione, e definiamolo all'interno di un modulo di nome Frazioni. Se ora volessimo accedere ad un campo di una variabile di tipo Frazione dichiarata in un altro modulo, in realtà non accediamo direttamente al campo in questione, ma **viene chiamata una particolare funzione** che per default accede in lettura o in scrittura al relativo campo.

Se per motivi di performance si vuole evitare la creazione delle funzioni di accesso e permettere anche a moduli esterni di accedere direttamente al campo si utilizza il modificatore di memorizzazione **explicit**:

```
type Frazione{
 explicit int num;
 explicit int den;
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
```

```
 end(){

 };
}
```

Ci sono situazioni in cui può far comodo utilizzare le funzioni di accesso per proteggere i dati memorizzati nei relativi campi. Per sovrascrivere le funzioni di accesso di default basta creare un blocco subito dopo il campo e quindi, se si vuole sovrascrivere la funzione di lettura, dichiarare una funzione di nome **get** mentre, per sovrascrivere la funzione di scrittura, dichiarare una funzione di tipo **set**. Queste funzioni devono però soddisfare le seguenti caratteristiche:

- La funzione get deve avere come tipo di ritorno il tipo del campo e non può avere parametri di input
- La funzione set deve avere void come "tipo di ritorno", e deve avere un solo parametro di input dello stesso tipo del campo

Entrambe le funzioni possono utilizzare this per accedere all'oggetto.

Ritorniamo al nostro tipo Frazione: una frazione deve avere necessariamente il denominatore diverso da 0 per esistere, ma così come è scritta potrebbe essere posto tranquillamente a 0 con il rischio di generare errori logici. Utilizziamo le funzioni di accesso per prevenire a questa eventualità:

```
type Frazione{
 int num;
 int den{
  void set(int val){
   if(val == 0)
    this.den = 1;
   else
    this.den = val;
  }
 };
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
  this.den = 1;
 };
 end(){

 };
}
```

Da notare che all'interno del modulo Frazioni ogni accesso ai campi di variabili Frazione sarà sempre diretto. Se si vuole che un accesso sia mediato dalla relativa funzione si deve utilizzare al posto del punto singolo (.) due punti affiancati (..)

```
var..den = 3;
```

Le funzioni di accesso sono molto potenti: negli esempi precedenti quando si dichiara un campo viene allocato lo spazio sufficiente per contenere il dato. In Meucci è possibile dichiarare campi **senza allocare spazio al campo stesso**, ma regolando gli accesso tramite le funzioni di accesso. Si crea quindi una sorta di campo fantasma, in cui viene data l'illusione dell'esistenza del campo quando gli accessi vengono tutti evasi dalle funzioni di accesso. Ovviamente per un campo ghost si dovranno specificare entrambe le funzioni di accesso.

Supponiamo di aggiungere un campo fantasma al nostro tipo Frazione che ritorna la parte intera del nostro oggetto (benché sarebbe più corretto definire una funzione distinta che calcoli la parte intera). Si utilizzerà il modificatore di memorizzazione **ghost** per creare un campo fantasma:

```
type Frazione{
 int num;
 int den{
  int get(){
   return this.den;
  }
  void set(int val){
   if(val == 0)
    this.den = 1;
   else
    this.den = val;
  }
 };
 ghost int parteIntera{
  int get(){
   return num/den;
  }
  void set(int v){

  }
 };
 init(int num, int de){
  this.num = num;
  this.den = de;
 };
 init(int n){
  this.num = n;
```

```
   this.den = 1;
  };
  end(){

  };
 }
```

Se un campo è ghost, allora anche all'interno del modulo l'accesso sarà mediato dalle funzioni.

5.3.1 Tipi explicit

Per migliorare le performance di accesso di un oggetto o per poter calcolare esattamente le dimensioni dello stesso si sarebbe tentati di dichiarare explicit ogni campo del tipo. In realtà vengono generate comunque altre strutture inutili (basta pensare alle vtable per gestire l'ereditarietà). Per impedire la creazione di queste altre strutture si può rendere l'intero tipo explicit ponendo tale parola chiave tra il nome e la parole type:

```
type explicit Struct{
```

In un tipo explicit l'accesso a tutti i campi sarà sempre diretto, e l'unico modificatore di memorizzazione ammesso è explicit (che viene naturalmente ignorato).

C'è però un prezzo da pagare: Un tipo explicit può estendere solo tipi explicit e per estendere un tipo explicit anche il sovrattipo dovrà essere explicit. Inoltre **non si potrà specificare un distruttore**. Questi argomenti verranno trattati nel capitolo dell'ereditarietà.

5.4 Accesso con parametri

A differenza di altri linguaggi di programmazione, Meucci permette di accedere ai campi anche attraverso dei parametri. Per comprendere meglio consideriamo il tipo IntArray, un array di interi. Per accedere ad un generico dato memorizzato nell'array si utilizza un parametro intero, che in questo specifico caso viene chiamato indice. Tramite le funzioni di accesso è possibile specificare parametri aggiuntivi per accedere al campo, che in Meucci deve possedere il modificatore **ghost**. Per specificare i parametri di accesso bisogna:

1. Scrivere subito dopo il campo in ordine i tipi dei parametri di accesso racchiusi tra parentesi quadre e separati da una virgola
2. Specificare entrambe le funzioni di accesso con i relativi parametri di accesso in ordine e sempre dopo l'eventuale parametro di scrittura per la funzione set.

Ecco lo pseudo-codice del nostro tipo IntArray:

```
type IntArray{
 int len;
 pt pointer;//Punta alla memoria dell'array
 ghost int elem[int]{
  int get(int index){
   if((index >=0) && (index < len)){
    'ritorna elemento'
   }
   else
    return 0;
  }
  void set(int val, int index){
   if((index >=0) && (index < len)){
    'setta elemento a val'
   }
  }
 };
 init(int len){
  'alloca spazio'
 };
 end(){
  'libera spazio'
 };
}
```

Per accedere al campo si devono specificare i parametri all'interno di parentesi quadre:

```
IntArray array = :new IntArray(5);
array.elem[3] = -34;
int v = array.elem[2 + 1]; // v == -34
```

5.5 Modificatori di accesso: read e shadow

Sempre per protezione, spesso si vuole impedire o limitare l'accesso ai campi da parte di moduli esterni. Per limitare l'accesso si utilizzano i seguenti modificatori di accesso:

- **shadow** Impedisce completamente l'accesso al campo ai moduli esterni. Specificando un campo shadow implicitamente lo si rende explicit, quindi specificare un campo sia shadow che con un modificatore di memorizzazione diverso da explicit genererà un errore del compilatore
- **read** Impedisce solo la scrittura al campo, non la lettura. Se il campo non è explicit, verrà generata solo la funzione di accesso in lettura, generando un errore nel caso si voglia creare anche una funzione di accesso in scrittura

Così il nostro tipo IntArray diverrà:

```
type IntArray{
 read int len;
 shadow pt pointer;//Punta alla memoria dell'array
 ghost int elem[int]{
  int get(int index){
   if((index >=0) && (index < len)){
    'ritorna elemento'
   }
   else
    return 0;
  }
  void set(int val, int index){
   if((index >=0) && (index < len)){
    'setta elemento a val'
   }
  }
 };
 init(int len){
  'alloca spazio'
 };
 end(){
  'libera spazio'
 };
}
```

5.6 Campi packed

Supponiamo di voler creare un tipo con 127 campi distinti di tipo real, abbiamo tre opzioni. La prima è di scrivere manualmente tutti i 127 campi cambiando il nome di ciascuno di essi, la seconda è di creare un array di 127 elementi, ma ciò creerebbe un overhead non trascurabile e una gestione della memoria difficoltosa, in quanto in Meucci non è presente il garbage collector come in Java. La terza opzione è di utilizzare la parola chiave **packed** seguita da 127, così il compilatore genererà automaticamente tutti i campi

```
type tti{
 real e packed 127;
}
```

L'accesso a ciascun campo viene effettuata come se avesse un parametro di tipo **uint**

```
var.e[u59] = 34.89;
```

Se si dichiara un campo packed, non sarà possibile assegnargli un modificatore di memorizzazione né inserire funzioni di accesso, il campo dovrà essere trattato come se fosse explicit.

Come vedremo nel prossimo capitolo, la lunghezza di un campo packed può essere un template di tipo numerico, rendendo questo strumento molto utile in varie circostanze.

5.7 Ereditarietà

Come nei linguaggi orientati ad oggetti, il linguaggio Meucci supporta il meccanismo dell'ereditarietà (**singola**). Per estendere un tipo basta inserire, dopo il nome del sovrattipo, la parola chiave **extends** seguito dal nome del sottotipo:

```
type sovraT extends sottoT{
...
}
```

Il sovrattipo erediterà tutti i dati del sottotipo, comprese le funzioni di accesso. Se il sovrattipo è creato in un modulo diverso da quello in cui è stato dichiarato il sottotipo, il modulo non potrà accedere ai membri shadow ereditati e potrà accedere solo in lettura ai campi read ereditati. In compenso, l'accesso ai dati ereditati sarà diretto, a meno che il campo non sia ghost o vengano usati i due punti.

5.7.1 Chiamata dei sovracostruttori

Quando si deve inizializzare un sovrattipo è necessario inizializzare anche i campi del sottotipo a cui non si avrebbe accesso (come i campi shadow). Per ovviare a ciò la prima istruzione di un costruttore derivato deve essere una chiamata ad un costruttore del sottotipo, utilizzando la pseudo-istruzione **super**:

```
...
init(){
 super(...);
 ...
}
...
```

5.7.2 Il modificatore override

La sovracclasse può modificare le funzioni di accesso dei campi ereditati sostituendole con altre più adatte. Per sovrascrivere le funzioni di accesso si riscrive nel sovrattipo il relativo campo con modificatore di memorizzazione **override**

e con l'eventuale modificatore di accesso e dei parametri di accesso, e quindi si scrivono le nuove funzioni di accesso che andranno a sostituire quelle vecchie.

Non è possibile effettuare l'override di campi explicit o shadow, e per i campi read è possibile modificare solo la funzione di lettura.

Capitolo 6

Template

Nel capitolo precedente abbiamo creato un tipo IntArray, un array di interi. Se ora volessimo creare un array di caratteri si dovrebbe creare un tipo CharArray, analogamente con ULongArray e così via. Si nota che la maggior parte del codice è lo stesso, cambia al massimo il tipo immagazzinato. Discorso analogo vale per le funzioni che operano su tali tipi.

Per questo è comodo creare non un solo tipo o funzione, ma una **famiglia di tipi e funzioni** con codice simile, in cui ciascun tipo o funzione è univocamente determinato da uno o più parametri. Una famiglia di tipi viene detta **tipo template**, analogamente si dice **funzione template**, mentre questi parametri di accesso sono detti **parametri template**. Non si possono creare allo stato attuale operazioni template.

Quandi si compila, questi tipi e moduli non vengomo memorizzati all'interno di un file .in, ma il compilatore genererà dei file .tin, generalmente più pesanti, che devono essere utilizzati come i normali file di importazione.

6.1 Parametri tipo e parametri numerici

Per creare un tipo o una funzione template bisogna far seguire al nome del tipo/funzione, tra parentesi quadre, tutti i parametri template utilizzati. Esistono due tipi di parametri template:

- Parametri **tipo**: il parametro può essere il nome di un tipo qualunque. Bisogna far precedere al nome del parametro la parola **typ**. Si possono aggiungere altre limitazioni ai valori ammissibili, poste dopo il nome del parametro: con **number** il tipo può essere solo un tipo intero (long, ushort, int, ...), mentre con **reference** può essere solo un tipo reference. Se si imposta la limitazione reference, si può obbligare il parametro ad assumere solo nomi di tipi che estendono un determinato tiporeference tramite la parola **extends** seguito dal sottotipo

- Parametri **numero**: il parametro può essere una qualunque costante **senza segno**. Bisogna far precedere al nome la parola chiave **num** e far seguire un numero che ne esprime il tipo (0 è ubyte, 1 è ushort, 2 è uint, 3 è ulong). Si possono aggiungere limitazioni ai valori che il parametro può assumere tramite i simboli $>$ e $<$ seguiti da costanti, che esprimono l'estremo inferiore e superiore del parametro.

Esempio:

```
type temp[typ T reference extends Frazione, num N 3 > 2 < 4]{
 init(){
 ...
 }
...
}
```

definisce per il tipo temp due parametri template: T parametro tipo reference che estende il tipo Frazione, N parametro numerico di tipo ulong maggiore di 2 e minore di 4 (può essere solo 3).

L'esempio seguente mostra come dichiarare e allonare memoria ad una variabile del tipo ottenuto dal tipo template temp con T = Intero (che estende il tipo Frazione semplicemente ponendo den=1) e N = 3:

```
temp[Intero, 3] var = :new temp[Intero, 3]();
//Le costanti utilizzate come parametri numerici sono
//sempre unsigned, anche se manca il prefisso u
```

6.2 Funzioni template

Ci sono inoltre delle **funzioni template**, ovvero funzioni definite dal compilatore che prendono come input parametri template e ritornano un parametro template:

- **SIZEOF** Accetta come parametro un parametro tipo e ritorna come parametro numerico la dimensione in byte della quantità di memoria allocata durante la dichiarazione
- **DIMENSION** Accetta un parametro tipo reference e ritorna come parametro numerico la dimensione in byte dell'oggetto
- **SUM** Accetta due o più costanti o parametri numerici e ritorna un parametro numerico che è la somma degli input
- **PROD** Accetta due o più costanti o parametri numerici e ritorna un parametro numerico che è il prodotto degli input

Quando viene utilizzato una funzione template, deve essere preceduta dal simbolo cancelletto `#` e i parametri devono essere racchiusi tra parentesi tonde e separati da virgole

```
uint vay = #SUM(u2, 3);
```

Appendice A

Array e StaticArray

Due tipi template molto utili e utilizzati sono **Array** (un array in Java) e **StaticArray** (un array in C), definiti entrambi nel modulo **Arrays** incluso di default in tutti i progetti. Il loro codice è molto utile per comprendere il funzionamento dei parametri template, per questo verrà riportato integralmente in questa appendice:

```
modulo Arrays depends Memory{
 type Array[typ T]{
  explicit read uint length;
  shadow pointer[T] memory;
  ghost T elem[uint]{
   T get(uint index){
    if(index < this.length){
     pointer[T] val=somma[T](this.memory, index);
     return val.el;
    }
    else{
     return (T) null; //Condizione di errore
    }
   }
   void set(T va, uint index){
    if(index < this.length){
     pointer[T] val=somma[T](this.memory, index);
     val.el=va;
    }
   }
  };
  init(uint size){
   pt memory=allocate(#SIZEOF(T)*size);
   this.memory=(pointer[T])memory;
```

```
   this.length=size;
  };
  end(){
   free((pt)this.memory);
  };
 }
 type explicit StaticArray[typ T, num L 2]{
  T elem packed L;
  init(){
   //Non e' necessario il costruttore,
   //in quanto l'allocazione della memoria
   //e' statica
  };
 }
}
```

```
Array[ubyte] b = :stack Array[ubyte](u2 + u1);
StaticArray[ubyte, 3] sb = :stack[ubyte, 3]();
b.elem[1] = u12b;
sb.elem[0] = b.elem[1] +u1B;
```

La differenza sta nel fatto che la dimensione del tipo Array si può determinare solo in fase di esecuzione, e quindi è necessario allocare dinamicamente la memoria, mentre la dimensione di StaticArray è determinata già in fase di compilazione (tramite il parametro template numerico L) e quindi il compilatore conosce già la dimensione dell'array.

```
type explicit pointer[typ TT]{
 TT el;
}
```

Il tipo template pointer[T] va inteso come un puntatore a elementi (non per forza oggetti) di tipo T.

Appendice B

Funzionalità future

Ecco un breve elenco delle funzionalità che forse verranno aggiunte nelle prossime versioni del compilatore

- Gestione eccezioni
- Possibilità di definire nuovi tipi primitivi
- Libreria multithreading
- Protocolli, ovvero astrazioni che permette il caricamento dinamico di moduli
- Possibilità all'interno di funzioni di accesso di chiamare le precedenti funzioni di accesso, al fine di migliorare la protezione dei dati membro