PARADIGMI DI PROGRAMMAZIONE

Corso A

Autore
Giuseppe Acocella
2024/25

Contents

1 OCaml

Descriviamo le caratteristiche dell'OCaml e forniamo snippet di codice relativi all'argomento trattato.

1.1 Dichiarazioni e tipi variabili

Variabili: Le variabili, una volta creato il binding ed assegnato un valore non è possibile modificarle.

Dichiarazione: La dichiarazione di una variabile avviene con la parola chiave **let** e può essere utilizzata per dichiarare ogni tipo di variabile, anche di tipo **funzione**.

Espressioni: Ogni riga di codice è un espressione, e mai un comando. Dunque essendo che stiamo aderendo al paradigma funzionale, non ci è permesso apportare modifiche ad **ambiente** e **memoria**. Le **espressioni** possono quindi contenere **bindings**.

```
1    let x = 5;;
```

1.1.1 Tipi, Cast ed Overload

Tipi: Enumeriamo i tipi base disponibili:

1. **int**: numeri interi
2. **float**: numeri frazionari *floating point* a doppia precisione
   (a) **Operatori**: $+.$ $-.$ $*.$ $/.$
3. **bool**: valori di verità (booleani): 'true' o 'false'
   (a) **Operatori**: && || *not*
4. **char**: singoli caratteri (racchiusi tra apici: ''a'')
5. **string**: stringhe (racchiuse tra virgolette: '"abcd"')
   (a) **Concatenazione**: $str_1 \wedge str_2$
   (b) **Overload**: Notiamo quindi che non verrà castato automaticamente un *int* a $float$, infatti esistono le operazioni specifiche per i float.
6. **unit**: tipo usato in casi particolari (simile a 'void' in altri linguaggi), prevede come unico valore '()'

Cast: In **OCaml** ogni tipo di cast è **esplicito**, dato che il sistema di tipi è molto **rigido**. Questo ci permette di poter sfruttare anche il **type inference** dell' **OCaml**, dato che nulla viene lasciato al caso.

```
1    let c = 'a' ;;
2    int_of_char c ;;
```

Questi tipi di cast esistono quindi da ogni e verso ogni tipo base.

1.2 Tuple e Liste

Mostriamo questi due tipi "composti", ognuno caratterizzato da specifiche particolari:

1. **Tuple**: Sono strutture non omogenee, definite anche n-uple, che variano in lunghezza, possono quindi contenere elementi di tipo diverso

```
1    let t = (10,"hello",12.5);;
2
```

2. **Liste**: Strutture omogenee, su cui è possibile effettuare varie operazioni di natura induttiva grazie alla ricorsione. Si mostrano queste operazioni nei capitoli successivi dedicati al **Pattern Matching**.

```
1    let numeri = [3; 5; -1; 9; 14; 21] ;;
2
```

1.3 Funzioni

Le **funzioni** sono il nucleo del paradigma funzionale. Ricordiamo che sono **espressioni** come tutte le altre e dunque possono essere legate ad un identificatore grazie ad un binding.

1. **Funzione Anonima**: Mostriamo una funzione nella sua essenza, prima ancora di effettuare il binding.

```
1    fun x -> x + 1 ;;
2
```

2. **Invocazione di Funzione**: Proprio come nel lambda calcolo possiamo invocare una funzione simile.

```
1    (fun x -> x + 1) 3;;
2
```

```
# (fun x -> x + 1) 10 ;;
- : int = 11
```

Dunque notiamo che questo tipo di notazione è fortemente influenzato dallo stile del **Lambda Calcolo**.

1.3.1 Currying e Zucchero Sintattico

Seguendo lo stile del **Lambda Calcolo** non è possibile avere più di un parametro in binding in una funzione. Questo stile è detto **Curryed**. Mostriamo quindi tutte le possibili rappresentazioni che l'**OCaml** offre:

1. **Stile Curryed**:

```
1    fun x -> fun y -> x+y ;;
2
```

2. **Stile Semplificato**:

```
1    fun x y -> x+y ;;
2
```

```
fun x y -> x+y;;
: int -> int -> int = <fun>
fun x -> fun y -> x+y;;
: int -> int -> int = <fun>
```

Figure 1: Notiamo che le due funzioni si equivalgono in entrambe le notazioni dato che hanno anche lo stesso tipo

1.3.2 Binding Funzione ad Identificatore

Il modo più comune per definire funzioni è quello di legarle ad un identificatore. Mostriamo due diversi modi di effettuare questo **binding**.

1. **Versione base**: Versione basata sulla notazione $->$

```
1    let somma = fun x y -> x+y ;;
2    somma 3 4 ;;
3
```

2. **Versione semplificata**: Versione basata sullo zucchero sintattico offerto dall'OCaml, questo stile è detto Curryed.

```
1    let somma x y = x+y ;;
2    somma 3 4;;
3
```

1.3.3 Notazione Curryed vs Tupla Parametri

Abbiamo già definito quale sia il modo più comune per definire funzioni, ossia lo stile **Curryed**. Ma nulla vieta l'utilizzo di una tupla per il passaggio dei parametri. Ci sono però delle differenze:

1. **Notazione Curryed Parametri**:

```
1    let somma x y = x+y ;;
2    somma 3 4;;
3
```

   Questa **permette** l'**applicazione parziale** di funzione, dato che nulla vieta il passaggio di un parametro singolo.

2. **Notazione Tupla Parametri**:

```
1    let somma (x, y) = x+y ;;
2    somma (3, 4);;
3
```

   Questa **vieta** l'**applicazione parziale** di funzione, dato che l'applicazione di funzione necessita un parametro che sia una tupla.

1.3.4 Funzioni Ausiliarie

Una funzione può anche essere visibile solo nello scope di una specifica **espressione** che la necessita. Mostriamo e descriviamo un esempio.

```
1    let sol a b c =
2    let delta =
3        b*.b -. 4.*.a*.c
4    in
5        ( (-.b +. sqrt delta) /. (2.*.a) , (-.b -. sqrt delta) /. (2.*.a) )
     ;;
6
```

1. **sol**: Funzione esterna che richiede tre parametri a, b, c.

2. **delta**: Funzione ausiliaria visibile solo nell'espressione a *riga 5*, che non prende argomenti, dato che utilizza come dati quelli contenuti nei *parametri formali* della funzione *sol*.

3. **Espressione eseguita**: Alla fine, si esegue il calcolo dell'espressione a *riga 5* che si basa su funzione ausiliare e su parametri della funzione esterna.

1.4 Controllo Flusso e Iterazione

Abbiamo la necessità di stabilire gli *if* per il controllo del flusso e successivamente anche come reiterare in questo linguaggio determinate espressioni.

Ricordiamo però che essendo questo linguaggio puramente funzionale (per ora), non ci è permesso modificare **ambiente** e **memoria**. Dunque non effettueremo iterazioni proprie ma invocheremo le funzioni ricorsivamente.

1. **if**: Mostriamo il controllo del flusso base.

```
1    let x = 23 ;;
2    let y = if x>10 then 1 else 0 ;;
3
```

2. **Ricorsione Base**: Mostriamo esempio sul Fibonacci.

```
1    let rec fibonacci n =
2    if n=0 || n=1 then n
3    else fibonacci (n-1) + fibonacci (n-2);;
4
```

Notiamo che in questo caso è necessario definire *fibonacci* come funzione ricorsiva *let rec*. Questo è zucchero sintattico che permette al compilatore di sapere che avrà bisogno di utilizzare una sorta di **combinatore Y** come abbiamo visto per il $\lambda -$ calcolo.

1.4.1 Funzioni Mutuamente Ricorsive

Mostriamo un semplice esempio di funzioni mutuamente ricorsive utilizzando il costrutto **match** che si approfondirà durante il **pattern matching**.

```
1    let rec pari n =
2    match n with
3    | 0 -> true
4    | x -> dispari (x-1)
5
6    and dispari n =
7    match n with
8    | 0 -> false
9    | x -> pari (x-1) ;;
```

1.5 Type Inference

Tutta la rigidità imposta sui tipi di permette di poter sfruttare un potentissimo **type inference**. Questo perchè **staticamente** il compilatore riesce a derivare quali siano i tipi dei parametri passati e il tipo della funzione stessa grazie a tutti i vincoli imposti precedentemente.

```
1    let sum_if_true test first second =
2    (if test first then first else 0)
3    + (if test second then second else 0);;
```

```
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

In questo caso ad esempio notiamo che il compilatore riesce perfettamente a capire di che tipo siano tutti i parametri e la funzione stessa prima ancora di valutarla definendoli. Questo per i seguenti motivi:

1. *test*: deve ritornare un booleano, dato che *if (test)*.
2. *first* e *second*: devono avere lo stesso tipo di 0 (quindi **int**) dato che non importa quale ramo dell'if si scelga a runtime, questi devono certamente arrivare allo stesso tipo.
3. *test*: se quindi *test* prendeva in input o *first* o *second* allora sappiamo inferire anche sul suo tipo in ingresso. Dunque il suo tipo complessivo dovrà essere:

$$(\ int -> bool\ )$$

1.5.1 Polimorfismo

A volte delle funzioni necessitano un tipo generico per stabilire cosa ritorneranno o cosa prenderanno come parametro. Un esempio è la funzione identità che prende un elemento di qualsiasi tipo e ritorna l'elemento stesso, il cui tipo corrisponde esattamente al tipo in partenza. Concetto simile al tipo **generics** in TypeScript.

```
val id : 'a -> 'a = <fun>
```

Figure 2: a' indica il tipo generico

1.6 Pattern Matching

Il Pattern Matching dell'OCaml è un operazione molto potente che permette di categorizzare tutte le tipologie dato un determinato dato in ingresso. Ha molti elementi in comune con un classico **switch** ma più ampio. Spesso infatti il **Pattern Matching** lo si utilizza sulla struttura di un dato complesso come liste e tuple.

Mostriamo un esempio su ricorsione, uno su destrutturazione di liste e uno sulla gestione delle tuple:

1. **Fibonacci Ricorsivo**:

```
let rec fibonacci n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fibonacci (n-1) + fibonacci (n-2);;
```

2. **Lunghezza lista**:

```
let rec lenList lst =
    match lst with
    | [] -> 0
    | a::lst1 -> 1 + lenList lst1;;
```

3. **Primo true nella tupla**:

```
let first_true t =
    match t with
    | (true,_,_) -> 1
    | (false,true,_) -> 2
    | (false,false,true) -> 3
    | (false,false,false) -> -1;;
```

Operazioni strutturali su liste Grazie al **Pattern Matching** riusciamo ad effettuare molte operazioni ricorsive basate sulla destrutturazione di liste. Ricordiamo però che le liste in OCaml sono realizzate come **Singly Linked List**, dunque manteniamo un riferimento alla testa della lista e passiamo ai successivi. Questo vuol dire che per concatenare due liste è necessario creare un nuovo puntatore, dato che se aggiungessimo semplicemente "in coda" alle liste già esistenti staremmo modificando la lista di partenza.

Esaustività Dato che l'OCaml non assume nulla, è necessario coprire la casistica per ogni possibile valore della variabile data in input al match. Questo non è obbligatorio ma se viene costruito un **Pattern Matching** non esaustivo viene sollevato un warning.

1.7 Funzioni Higher Order

Abbiamo mostrato come sia possibile effettuare ricorsione strutturale su liste con il **Pattern Matching**. Spesso però eseguiremo operazioni molto simili in funzioni diverse. Infatti molte volte ci troveremo ad esempio a "scorrere tutti gli elementi" oppure "applicare una funzione a tutti", o ancora "filtrare in base a $predicato_x$". Possiamo quindi stabilire delle funzioni che acquisiscono funzioni/predicati e liste su cui vogliamo che operino. In questo modo avremo una disponibilità di "operazioni macro" a cui fare riferimento. Elenchiamone alcune:

1. **Map**:

```
let rec map f lis =
    match lis with
    | [] -> []
    | x::lis1 -> f x::map f lis1 ;;
```

2. **Filter**:

```
let rec filter p lis =
    match lis with
    | [] -> []
    | x::lis1 -> if p x then x::filter p lis1
                 else filter p lis1 ;;
```

3. **For All**:

```
let rec forall p lis =
    match lis with
    | [] -> true
    | x::lis1 -> if p x then forall p lis1
                 else false ;;
```

4. **Fold (from) Right**:

```
let rec fold_right f lis a =
    match lis with
    | [] -> a
    | x::lis1 -> f x (fold_right f lis1 a) ;;
```

Esempio di Applicazione Mostriamo un esempio di applicazione di somma di tutti gli elementi di una lista con l'utilizzo della **Fold Right**.

```
    let somma lis = fold_right (+) lis 0 ;;
    somma [3;2;4] ;;
```

1.8 Tipi Custom

Rappresentiamo come definire nuovi tipi custom.

1.8.1 Record e Definizione Type

Una delle operazioni principali sui tipi è quella di stabilirne uno nuovo:

```
1    type data = int*int*int ;;
```

In questo modo stiamo quindi definendo un nuovo tipo come tupla, sapendo che le tuple si basano sul prodotto cartesiano di insiemi.

Record Questo tipo non è nient'altro che una maniera più avanzata di utilizzo del prodotto cartesiano di insiemi.

```
1    type punto_2d = { x: float; y: float; } ;;
2    let p = { x = 3.; y = -4. } ;;
```

Elenchiamo un po' di caratteristiche di questa definizione di tipo:

1. I Record sono **immutabili**, dunque sono rappresentati come oggetti ma non equivalgono a questi ultimi.

2. Non esiste nessun metodo, possiamo si assegnare delle funzioni ai campi del record, ma queste ultime non vedranno gli altri campi del record in questione dato che non esiste nessun costrutto simile al **this**.

3. La **dot notation** permette un accesso diretto ai campi, quindi un record non ha la necessità di mantenere la "posizionalità" dei campi.

4. il **functional updating** non è nient'altro che la creazione di un nuovo record partendo da uno già esistente e modificando alcuni dei suoi campi.

```
1    type persona = {nome: string; cognome: string; via: string; citta:
     string} ;;
2
3    let padre = {nome:"Mario" ; cognome:"Rossi" ; via:"Via Bianchi" ;
     citta:"Roma"} ;;
4    let figlia = { padre with nome = "Bianca" } ;;
```

1.8.2 Tipi Unione/Somma - Variant

Grazie ai tipi **variant** è possibile generare un nuovo **tipo** come **unione** di tipo già esistenti.

1. **TypeScript**:

```
type t = string | int
```

2. **OCaml**:

```
type numero_testo =
    | Txt of string
    | Num of int ;;
```

Notiamo delle differenze, infatti l'**OCaml** richiede un **etichetta** detta anche **costruttore**, che segnali di quale "partizione" faccia parte nel nuovo tipo unione definito.

Tipi Strutturali Ricorsivi Possiamo quindi definire le **liste** come caso particolare di tipo **union ricorsivo**.

```
type lista_di_int =
    | Nil
    | Elem of int * lista_di_int ;;

let lst = Elem (3 ,Elem (4, Elem (6,Nil))) ;;
```

1.8.3 Tipi Enumerazione

Anche in **OCaml** è possibile definire le **enum**, utilizzando solo **costruttori** è possibile infatti raggiungere lo stesso significato di una normale **enum**.

```
type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom ;;
```

Questo ci permette quindi di eseguire controlli sul tipo grazie al **Pattern Matching**:

```
let is_weekend g =
    match g with
    | Sab | Dom -> true
    | _ -> false ;;
```

1.8.4 Tipi Opzione

I tipi **opzione** sono particolari tipi **variant** che consentono di stabilire se esiste o meno un dato grazie ai costruttori *Some* e *None*. In questo modo possiamo gestire il ritorno di nessun valore.

```
type A option =
    | Some of A
    | None ;;
```

1.9 Cenni Programmazione Imperativa

In OCaml è anche possibile scrivere codice secondo il paradigma imperativo, abbiamo però bisogno di nuovi tipi di dati che ci permettano di aggiornare e variare il loro contenuto. Ricordiamo però che l'utilizzo del paradigma imperativo richiede delle regole di "buon uso" che permettono di non "distruggere" tutta la rigidità costruita fino ad ora grazie al paradigma funzionale. Elenchiamo alcune buone norme:

1. Si preferisce l'utilizzo del funzionale all'imperativo.
2. Ogni funzione imperativa, se scritta, deve essere incapsulata per fare in modo che il suo stato non influenzi le funzioni "indipendenti" definite nel resto del codice.

1.9.1 Tipi di dato

Elenchiamo i tipi di dati:

1. **Variabili (Riferimenti)**: Abbiamo la necessità di stabilire nuove variabili che siano modificabili. Mostriamone la sintassi:
   (a) **Dichiarazione**:

```
let x = ref 12 ;;
```

```
val x : int ref = {contents = 12}
```

   (b) **Accesso a Contents**:

```
x.contents ;;
x.contents <- 13;;
```

   (c) **Sintassi propria di Accesso e Assegnamento**:

```
!x ;;
x := 14 ;;
```

   Ricordiamo che l'accesso a **Contents** del punto b e le operazioni di **Accesso** e **Assegnamento** sono identiche.

2. **Array**: Gli array sono modificabili, a differenza delle liste fino ad ora osservate. Mostriamo la loro sintassi:

```
let a = [|3;5;2|] ;;
let n = Array.length a;;
let e = a.(1) ;;
a.(1) <- 6 ;;
```

   In ordine abbiamo rappresentato **dichiarazione e inizializzazione**, **utilizzo di metodo**, **accesso a campo** e **assegnamento a campo**.

3. **Record Mutabili**: Possiamo definire dei tipi **record** e successivamente stabilire quali campi vogliamo che siano **variabili**:

```
type persona =
{
    nome: string;
    cognome: string;
    mutable eta: int;
}
```

4. **Eccezioni**: Possiamo gestire gli errori come negli altri linguaggi, **OCaml** offre quindi la possibilità di sollevare eccezioni, proprio come il *throw* in JS. Mostriamo degli esempi:

```
exception Lista_vuota ;;
exception Stringa_errata of string ;;
raise Lista_vuota;;
failwith "messaggio di errore" ;;
```

1.9.2 Comandi Iterativi

Dato che abbiamo mostrato come sia possibile aggiornare un valore di una variabile, ora possiamo iterare aggiornando degli indici, non essendo più costretti ad effettuare ricorsione.

1. **While**:

```
let x=ref 10 in
while !x>0 do
    print_endline (string_of_int !x) ;
    x := !x/2
done
```

2. **For**:

```
for i = 1 to 10 do
    print_endline (string_of_int i)
done ;;
```

2 Macchine Astratte e Interpreti

Tutte le macchine attuali si ispirano ad un antico modello, detto **modello di Von Neumann**. Questo prevedeva l'interazione tra due elementi principali, ossia:

1. **Unità centrale di elaborazione**
2. **Memoria**

Grazie a questo schema le macchine di oggi riescono ad eseguire tutte le operazioni che gli assegniamo.

Questa esecuzione di operazioni avviene a sua volta secondo uno schema preciso:

Ciclo Fetch, Decode, Execute Grazie a questo ciclo, le istruzioni vengono prima caricate dalla memoria **Fase Fetch**, successivamente vengono tradotte da linguaggio macchina in binario **Fase Decode** e successivamente vengono eseguite **Fase Execute**.

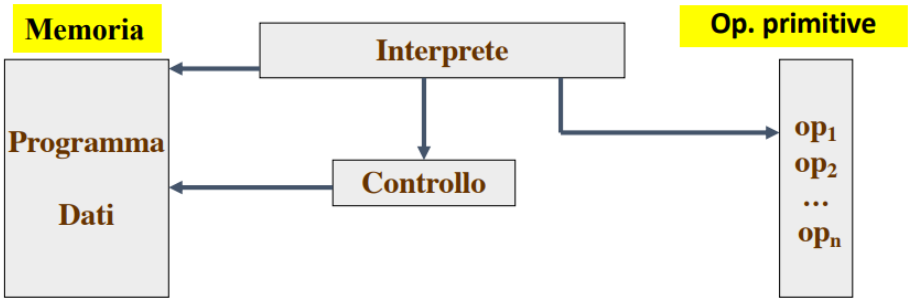
2.1 Tipi di linguaggi

Un linguaggio dunque per essere elaborato da una macchina deve essere convertito in linguaggio macchina. Esistono 3 tipi di approccio:

1. **Compilato**: Viene compilato tutto il sorgente e vengono eseguiti dei controlli a tempo statico. Dunque tutto il sorgente viene compilato e reso eseguibile.
2. **Interpretato**: L'interpretazione esegue "riga per riga" i comandi del sorgente. In questo caso non è quindi possibile eseguire dei controlli a tempo statico.
3. **Misto**: Questa metodologia utilizza un compilatore intermedio che compila il **source** in **bytecode**. Successivamente viene dato il **bytecode** ad un interprete che lo esegue riga per riga.

2.2 Macchina Astratta

Una Macchina Astratta esegue comandi step by step, omettendo i dettagli più a basso livello.



2.2.1 Macchina Astratta e Linguaggi

Elenchiamo tutti gli elementi necessari alla realizzazione di una macchina astratta di un linguaggio su una macchina host:

1. **L**: Linguaggio ad alto livello
2. $\mathbf{M}_L$: Macchina astratta del linguaggio L
3. $\mathbf{M}_H$: Macchina fisica host (ospite)

Questi elementi interagiscono tra di loro in modo diverso in base a che tipo di traduzione scegliamo tra linguaggio ad alto livello L e macchina fisica M_H.

1. **Interprete Puro**: La macchina astratta M_L permette la traduzione riga per riga del linguaggio L per essere eseguito dalla macchina host M_H.
2. **Compilatore Puro**: Non esiste una vera e propria macchina astratta M_L, infatti viene direttamente tradotto tutto il sorgente L in linguaggio macchina di M_H.
3. **Misto**: Questo tipo di approccio si appoggia su una prima fase di compilazione da sorgente L in linguaggio di una macchina astratta intermedia M_L che interpreterà passo passo il compilato in "bytecode".

2.2.2 RTS - Run Time Support

Durante il **tempo di esecuzione** è necessario fornire supporto al programma che avrà bisogno di un contesto per poter runnare in maniera solida. Il ruolo dell'**RTS** quindi è quello di fornire supporto al programma main. Nel linguaggio C ad esempio si occupa della chiusura/apertura dei record di attivazione e anche della gestione dello stack e dell'heap.

2.2.3 Fasi Front/Back End

Elenchiamo delle macrofasi della compilazione/interpretazione:

1. **Front End**: Il front end della compilazione si occupa di tutti gli aspetti del linguaggio ad alto livello, quindi il suo scanning, parsing e gestione semantica.
   (a) **Scanning**: Si acquisisce tutto il sorgente e lo si scannerizza per definirne tutti i token. Questo dunque produrrà una lista di tanti token diversi.
   (b) **Parsing**: Data la lista di token, si verifica in questa fase se sia o meno possibile formare quella sequenza di token data una specifica grammatica del linguaggio in questione. Si produce quindi un **AST**, ossia un Abstract Syntax Tree.
   (c) **Semantica**: Dato un **AST** possiamo quindi stabilire il suo significato. Elenchiamo le possibili sottofasi:
      i. **Type Checking**: Questo controllo viene eseguito staticamente nei linguaggi compilati. Questo permette di escludere tutti gli errori causati da interazioni tra tipi non compatibili. Nei linguaggi interpretati come JS, non viene effettuata un analisi statica di tipi, infatti ogni variabile si porta dietro un **descrittore** che contiene tutte le sue caratteristiche di tipo.

Dunque durante l'esecuzione, il confronto tra descrittori permette di segnalare potenziali errori di tipo. Questo approccio non è per nulla efficiente e rigido, dato che dipendera quasi esclusivamente dall'andamento del programma.

ii. **Ottimizzazioni Statiche**: Vengono effettuati dei controlli, e dove possibile, il compilatore ottimizza alcune operazioni. Alcuni linguaggi ad esempio aggiungono in questa fase la valutazione cortocircuitata del OR o dell'AND.

2. **Back End**: Questa fase traduce effettivamente il sorgente **L** in linguaggio della macchina host M_H.

2.3 Ambiente e Tipi di Scoping

Un ambiente è una **funzione** che dato un **Id** ritorna un **valore**, dato che in questo caso stiamo facendo riferimento a dei linguaggi funzionali che non si avvalgono di nessuna funzione che copra il ruolo della **memoria**, essendo tutti gli identificatori costanti.

Nell'implementazione del MiniCaml mostreremo un ambiente Σ.

Possiamo quindi dire che lo **scopo** dell'**ambiente** è quello di tenere conto di tutti i **binding** tra nome - entità.

Visibilità Identificatori L'ambiente permette quindi di stabilire la visibilità degli identificatori, in relazione alla loro "locazione" nel sorgente.

2.3.1 Scoping Statico vs Dinamico

Descriviamo le caratteristiche fondamentali che delineano la tipologia di **scoping**:

1. **Statico**: A tempo di dichiarazione, la funzione si porta dietro uno "snapshot" dell' ambiente corrente, chiamata **chiusura**. Questa viene utilizzata quando occorrono nel corpo della funzione delle variabili libere, queste ultime infatti verranno "chiuse" con l'ambiente a tempo di dichiarazione della funzione stessa.

2. **Dinamico**: A tempo di dichiarazione non viene prodotta **nessuna chiusura**. In questo modo, quando occorreranno delle variabili libere nel corpo della funzione, queste faranno riferimento alle occorrenze più recenti presenti nell'ambiente generale, e non nella chiusura, di cui abbiamo stabilito la non esistenza.

Blocchi e Shadowing Un blocco solitamente è la parte di codice contenuto in parentesi graffe, ma non sempre è così, infatti in OCaml il blocco si formava durante l'utilizzo del costrutto *in*. In ogni caso bisogna stabilire cosa occorre fare in caso di ridichiarazione di identificatore in un blocco, ci sono due possibili interpretazioni:

1. **Shadowing**: In questo caso, l'originale valore dell'identificatore viene coperto da quello più recente redichiarato nel blocco.

2. **Sollevare Eccezione**: Alcuni linguaggi non vogliono permettere la redichiarazione di identificatori per poter assicurare maggiore sicurezza, il **Java** ad esempio non permette la redichiarazione di identificatori nei blocchi più interni, forzando gli utenti ad utilizzare sempre nuovi identificatori.

2.3.2 Record di attivazione e Stack

Possiamo denotare quali siano le differenze sulla realizzazione dello stack tra scoping statico e dinamico. Elenchiamo i vari campi di un record di attivazione:

1. **Parametri**
2. **Variabili Locali**
3. **Dynamic Link** (ossia da dove sono stato invocato)
4. **Static Link** (ossia dove sono stato dichiarato)
5. **Indirizzo di Ritorno**

Dunque in uno stack di un linguaggio basato su **scoping dinamico**, il campo relativo allo **Static Link non è presente**. In questo modo non sarà possibile catturare nessuna chiusura di ambiente a tempo di dichiarazione.

Chiaramente in alcuni casi **Static Link** e **Dynamic Link** sono correlati ed in altri no. Invece ad esempio in **C**, non avendo la possibilità di annidare dichiarazioni di funzioni, tutti gli **Static Link** faranno riferimento allo stesso, ossia all'ambiente globale.

La gestione dello stack dei record di attivazione solitamente è affidata all'Run Time Support, che si occupa di ritornare al chiamante, spostando il program counter.

3 MiniCaml

Il MiniCaml è una versione ridotta dell'OCaml, di cui mostreremo l'intera implementazione, iniziando da i tipi, fino all'evalutazione delle intere espressioni.

3.1 Gestione Formale Ambiente Sigma

L'ambiente sarà una lista di associazioni, partendo da questi concetti:

1. **Definizione Formale Ambiente**: Definiamo formalmente l'ambiente.

$$\Sigma \to \text{Val} + \text{Unbound}$$

2. **Estensione Ambiente**: Assumiamo di poter aggiungere nuovi binding in questo modo:

$$\Sigma\,[x = z](y) = \begin{cases} v & \text{se} \quad x = y \\ \Sigma(y) & \text{altrimenti} \end{cases}$$

Una volta definita in maniera formale il nostro ambiente, risulta necessario fornire delle implementazioni in OCaml.

3.2 Implementazione Ambiente - Tipo Funzione env

Definiamo tutti gli elementi necessari:

1. **Tipo - Ambiente Polimorfo**:

```
type t1 env = ide -> t1
```

2. **Dereferencing in ambiente s con input x**:

```
s x
```

3. **Ambiente iniziale vuoto**:

```
let emptyenv = fun x -> UnBound
```

4. **Estensione Ambiente con nuovo binding**:

```
let bind (s: evT env) (x: ide) (v: evT) =
function (i: ide) -> if (i = x) then v else (s i)
```

   Qui notiamo che la funzione bind prende 3 parametri, ossia s (ambiente), x (identificatore) e v (valore valutabile). Di conseguenza bind torna una funzione che:

   (a) Se già esiste il bind sulla x allora lo copriamo con il valore di v corrente.

   (b) Altrimenti torniamo semplicemente la funzione che associa il valore di i in s.

3.3 Categorie Sintattiche - Tipi Opzione

Bisogna adesso definire tutte le categorie sintattiche come tipi, utilizzando i tipi opzione come livello di astrazione. Dunque è necessario immaginare le etichette dei tipi opzione come l'effettiva categoria sintattica (alto livello), mentre il tipo lato OCaml sarà a destra del costrutto *of* (basso livello).

1. **Identificatori**: Nel linguaggio MiniCaml saranno riconosciuti con *ide*, ma a livello di OCaml sono stringhe.

```
type ide = string;;
```

2. **Tipi degli Identificatori**: Anche in questo caso risulta necessario stabilire un tipo OCaml che copra i tipi del MiniCaml.

```
type tname =  TInt | TBool | TString | TClosure | TRecClosure |
TUnBound
```

3. **Espressioni**: In OCaml vedremo le categorie sintattiche del MiniCaml come tipi opzione con relative etichette e tipi reali.

```
type exp =
| EInt of int
| CstTrue
| CstFalse
| EString of string
| Den of ide
(* Operatori binari da interi a interi*)
| Sum of exp * exp
| Diff of exp * exp
| Prod of exp * exp
| Div of exp * exp
(* Operatori da interi a booleani *)
| IsZero of exp
| Eq of exp * exp
| LessThan of exp*exp
| GreaterThan of exp*exp
(* Operatori su booleani *)
| And of exp*exp
| Or of exp*exp
| Not of exp
(* Controllo del flusso, funzioni *)
| IfThenElse of exp * exp * exp
| Let of ide * exp * exp
| Letrec of ide * ide  * exp * exp
| Fun of ide * exp
| Apply of exp * exp
```

Categoria sintattica delle espressioni.

4. **Tipi Evalutazione**:

```
type evT =
| Int of int
| Bool of bool
| String of string
| Closure of ide * exp * evT env
| RecClosure of ide * ide * exp * evT env
| UnBound
```

Questo tipo il tipo di dato restituito dalla funzione principale **eval**, dopo averla invocata su una **exp** e su un corrente ambiente di tipo **evT env**.

3.4 Type Checking Dinamico con Descrittori

Assumiamo di voler effettuare dei controlli di tipo a tempo dinamico. Di conseguenza avremo bisogno di descrittori di tipo, con i quali effettueremo i necessari confronti per poter stabilire la correttezza dei tipi. Mostriamo le due funzioni che effettueranno questi controlli a pagina successiva.

1. **Funzione che associa valore a relativo descrittore**:

```
let getType (x: evT) : tname =
match x with
| Int(n) -> TInt
| Bool(b) -> TBool
| String(s) -> TString
| Closure(i,e,en) -> TClosure
| RecClosure(i,j,e,en) -> TRecClosure
| UnBound -> TUnBound
```

Osservando i tipi delle espressioni valutabili, notiamo che ognuno di questi si portava dietro due informazioni, ossia il dato effettivo e il suo descrittore. La funzione **getType** grazie al **pattern matching** associa il formato *costruttore*(*dato*) al tipo corrispondente in OCaml.

2. **Funzione che associa valore a relativo descrittore**:

```
let typecheck ((x, y) : (tname*evT)) =
match x with
| TInt -> (match y with
           | Int(u) -> true
           | _ -> false
           )
| TBool -> (match y with
            | Bool(u) -> true
            | _ -> false
            )
| TString -> (match y with
              | String(u) -> true
              | _ -> false
              )
| TClosure -> (match y with
                | Closure(i,e,n) -> true
                | _ -> false
                )
| TRecClosure -> (match y with
                  | RecClosure(i,j,e,n) -> true
                  | _ -> false
                  )
|TUnBound -> (match y with
              | UnBound -> true
              | _ -> false
              )
```

La funzione **typecheck** acquisisce una tupla come parametro, dove la tupla è una coppia nel formato (*NomeTipo* , *EspressioneEvalutabile*), ed esegue in ordine:

(a) Pattern matching su x, associandolo ad un possibile *NomeTipo*.

(b) Pattern matching su y, verificando se il *NomeTipo* dato da x corrisponde o meno al tipo dato dal formato *Costruttore*(*Dato*) delle espressioni evalutabili.

3.4.1 Operazioni di Base

Elenchiamo tutte le operazioni di base espresse come funzioni vere e proprie. Notiamo che tutte queste effettueranno un controllo dinamico sui tipi prima di eseguire effettivamente l'operazione, tutto questo grazie al pattern matching.

```
1   (* Somma fra interi *)
2   let int_plus(x, y) =
3       match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
4       | (true, true, Int(v), Int(w)) -> Int(v + w)
5       | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
6
7   (* Differenza fra interi *)
8   let int_sub(x, y) =
9       match(typecheck(TInt,x), typecheck(TInt,y), x, y) with
10      | (true, true, Int(v), Int(w)) -> Int(v - w)
11      | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
12
13  (* Operazioni logiche *)
14  let bool_or(x,y) =
15      match (typecheck(TBool,x), typecheck(TBool,y), x, y) with
16      | (true, true, Bool(v), Bool(w)) -> Bool(v || w)
17      | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
18
19  let bool_not(x) =
20      match (typecheck(TBool,x), x) with
21      | (true, Bool(v)) -> Bool(not(v))
22      | (_,_) -> raise ( RuntimeError "Wrong type")
23
24  (* Operazioni di confronto *)
25  let less_than(x, y) =
26      match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
27      | (true, true, Int(v), Int(w)) -> Bool(v < w)
28      | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
29
30  let greater_than(x, y) =
31      match (typecheck(TInt,x), typecheck(TInt,y), x, y) with
32      | (true, true, Int(v), Int(w)) -> Bool(v > w)
33      | (_,_,_,_) -> raise ( RuntimeError "Wrong type")
```

Anche se non corrisponde alla lista completa di operazioni base eseguibili, notiamo il loro pattern:

1. Acquisiscono un numero di parametri in base alla loro arietà.
2. Eseguono pattern matching sui parametri.
3. Si assicurano che il typecheck sia vero, invocando la funzione **typecheck** con i correnti parametri e i descrittori attesi.
4. Se il typecheck ha avuto successo, eseguono l'operazione a basso livello (livello OCaml).

3.5 Evalutazione - Semantica Funzione eval

```
let rec eval (e:exp) (s:evT env) : evT =
    match e with
    // 1. Tipi base: tipo exp -> tipo evt
    | EInt(n) -> Int(n)
    | CstTrue -> Bool(true)
    | CstFalse -> Bool(false)
    | EString(s) -> String(s)
    | Den(i) -> (s i)

    // 2. Invocazione operazioni base con valutazione Eager
    | Prod(e1,e2) -> int_times((eval e1 s), (eval e2 s))
    | Sum(e1, e2) -> int_plus((eval e1 s), (eval e2 s))
    | Diff(e1, e2) -> int_sub((eval e1 s), (eval e2 s))
    | Div(e1, e2) -> int_div((eval e1 s), (eval e2 s))
    | IsZero(e1) -> is_zero (eval e1 s)
    | Eq(e1, e2) -> int_eq((eval e1 s), (eval e2 s))
    | LessThan(e1, e2) -> less_than((eval e1 s),(eval e2 s))
    | GreaterThan(e1, e2) -> greater_than((eval e1 s),(eval e2 s))
    | And(e1, e2) -> bool_and((eval e1 s),(eval e2 s))
    | Or(e1, e2) -> bool_or((eval e1 s),(eval e2 s))
    | Not(e1) -> bool_not(eval e1 s)

    // 3. Controllo del flusso
    | IfThenElse(e1,e2,e3) ->
        let g = eval e1 s in
            (match (typecheck(TBool,g),g) with
            |(true, Bool(true)) -> eval e2 s
            |(true, Bool(false)) -> eval e3 s
            |(_,_) -> raise ( RuntimeError "Wrong type")
            )

    // 4. Gestione Blocco di Scoping
    | Let(i, e, ebody) -> eval ebody (bind s i (eval e s))

    // 5. Utilities per funzioni, da definizioni a invocazioni
    | Fun(arg, ebody) -> Closure(arg,ebody,s)
    | Letrec(f, arg, fBody, leBody) ->
        let benv = bind (s) (f) (RecClosure(f, arg, fBody,s)) in
            eval leBody benv
    | Apply(eF, eArg) ->
        let fclosure = eval eF s in
            (match fclosure with
            | Closure(arg, fbody, fDecEnv) ->
                let aVal = eval eArg s in
                let aenv = bind fDecEnv arg aVal in
                eval fbody aenv
            | RecClosure(f, arg, fbody, fDecEnv) ->
                let aVal = eval eArg s in
                let rEnv = bind fDecEnv f fclosure in
                let aenv = bind rEnv arg aVal in
                eval fbody aenv
            | _ -> raise ( RuntimeError "Wrong type")
            )
```

Commentiamo punto per punto lo sviluppo di questa funzione a pagina successiva.

La funzione eval esegue pattern matching sul parametro e passato di tipo **exp** in un ambiente s di tipo **evT** e ritorna un **evT**. In base al caso trovato esegue specifiche operazioni:

1. **Tipi Base**: Se avviene **match** con un **tipo base**, allora sarà necessario ritornare il corrispettivo dato in formato **evalutabile**, come richiesto anche dal tipo di ritorno della **eval**.
2. **Operazioni Base**: Se avviene **match** con un **operazione base**, la funzione eval esegue:
   (a) Una **evalutazione** dei **parametri** prima dell'invocazione dell'effettiva operazione base. Ricordiamo che questo tipo di approccio è denominato **Eager**, dato che evalutiamo i parametri prima di invocare una funzione.
   (b) Quando i parametri sono stati **evalutati completamente**, si invocano le funzioni delle **operazioni base**.
3. **Controllo del Flusso**: Se avviene **match** con un **controllo del flusso**, la funzione eval esegue:
   (a) Un evalutazione del primo parametro, ossia la guardia, ponendola in una variabile s e creando un binding in un nuovo blocco.
   (b) Dato il nuovo blocco in cui è visualizzabile l'esito della guardia, matcha l'invocazione del typecheck sull'esito della guardia, mentre il secondo parametro (sempre la guardia) ci indicherà il ramo dell'if da visitare. Dunque evaluta $e2$ oppure $e3$ in base a se la guardia fosse $true$ o $false$.
4. **Gestione scoping e nuovo blocco**: Se avviene **match** con una **creazione di nuovo blocco**, la funzione eval esegue:
   (a) *Ebody* è il corpo del blocco in cui voglio rendere visibile il nuovo identificatore, dunque l'invocazione dell'*eval* viene fatta su *ebody* come parametro *exp* ma abbiamo bisogno di destrutturare la dichiarazione.
   (b) Valuto l'espressione che sto associando all'identificatore, ed invoco la funzione *bind* sul corrente ambiente s, l'identificatore da legare i e l'esito dell'*eval* sulla right hand della dichiarazione.
5. **Utilities sulle funzioni**: Se avviene **match** con un **invocazione o dichiarazione di funzione**, la funzione eval esegue:
   (a) **Dichiarazione di Funzione Semplice**: Viene restituita una tripla di tipo *Closure* che si porta dietro tre argomenti, ossia formali, corpo della funzione e ambiente a tempo di dichiarazione.
   (b) **Dichiarazione di Funzione Ricorsiva**: Data una quadrupla contenente l'identificatore della funzione, i formali, il corpo della funzione ed il corpo della dichiarazione, crea un blocco in cui è visibile *benv*, ossia l'estensione del corrente ambiente con l'associazione tra la funzione e la sua quadrupla e restituisce l'invocazione dell'*eval* sul corpo della dichiarazione nell'ambiente esteso *benv*.

6. **Invocazione di funzione**: Abbiamo due sottocasi:

   (a) **Invocazione di funzione semplice**:

      i. Rende visibile $aVal$, ossia l'evalutazione degli attuali nel corrente ambiente s.

      ii. Effettua il *bind* tra i formali e $aVal$, estendendo la chiusura. Questa estensione viene resa visibile nel blocco successivo con il nome di $aenv$.

      iii. Evaluta il corpo della funzione $fbody$ nell'ambiente $aenv$.

   (b) **Invocazione di funzione ricorsiva**:

      i. Rende visibile $aVal$, ossia l'evalutazione degli attuali nel corrente ambiente s.

      ii. Estende la chiusura con l'identificatore f e la chiusura della funzione f stessa. Produce quindi un $rEnv$ visibile nel blocco successivo.

      iii. Esegue il bind tra formali e evalutazione degli attuali ($aVal$) nell'ambiente reso visibile nello step precedente, ossia $rEnv$, rendendo visibile un nuovo ambiente $aEnv$.

      iv. Evaluta il corpo della funzione $fBody$ nell'ambiente $aEnv$.

4 Paradigma Orientato agli Oggetti

In questo capitolo si discuteranno le caratteristiche della programmazione orientata agli oggetti e si mostreranno degli esempi pratici su linguaggio **Java**.

Potremmo immaginare il paradigma **object oriented** come un evoluzione

dell'imperativo. La caratteristica di questo paradigma è infatti **incapsulare** lo stato del programma in diverse entità chiamate **oggetti** che possono comunicare tra di loro solo grazie ai **metodi** che espongono.

Oggetti e Astrazione Un concetto fondamentale di questo paradigma è che ogni oggetto non ha la necessità di conoscere lo stato degli altri oggetti. Questo ci permette di creare un vero e proprio livello di astrazione superiore, rendendo più modulare il codice sorgente. Se infatti modificassimo la logica grazie alla quale si mantiene uno stato di un oggetto, ma non modificassimo i suoi metodi esposti, allora gli oggetti esterni non dovranno variare il modo in cui comunicavano con quest'ultimo.

4.1 Proprietà Principali del OOP

Elenchiamo tutte le proprietà caratteristiche della Programmazione Orientata agli Oggetti:

1. **Incapsulamento**: Ogni oggetto mantiene il proprio stato privato, permettendone la modifica solo attraverso specifici metodi, esponendone in maniera protetta l'accesso. Idealmente lo stato di ogni oggetto non dovrebbe essere accessibile agli altri (Information Hiding). In questo modo è possibile frazionare lo stato globale del programma in tanti stati di oggetti.
2. **Ereditarietà**: Ogni classe estende una superclasse, ereditando campi e metodi.
3. **Polimorfismo**: Una classe non limita l'utilizzo dei suoi metodi ad un tipo specifico, ma si stabilisce un tipo generico che fa da placeholder ad un tipo concreto. Dunque il tipo sarà generico ma anche omogeneo.
4. **Interfacce**: Mostra una "specifica" di campi e metodi che si aspetta da una classe. La classe dovrà implementare i metodi dell'interfaccia. Dunque mostra cosa una classe offre ed espone all'esterno.
5. **Principio di Sostituzione**: Principio per cui un oggetto può essere utilizzato al posto di un altro in maniera controllata. Assumiamo che una classe $Cube$ sia estensione della classe $Shape$. Ovunque sia richiesta un istanza di $Shape$ è possibile fornire un istanza di $Cube$, staremmo semplicemente utilizzando qualcosa "di più specifico" senza una particolare motivazione, ma funzionerebbe per il principio di sostituzione.

OOP vs Imperativo e Funzionale Ricordiamo che l'imperativo era caratterizzato da sequenze di comandi che influivano sullo stato del programma, mentre il funzionale si basa su una composizione di funzioni. A differenza di questi due paradigmi, l'object oriented si basa su stati incapsulati in oggetti ed interazioni tra gli oggetti stessi tramite metodi.

4.2 Object Based vs Class Based

Analizziamo due stili di programmazione orientata agli oggetti:

1. **Object Based**: Stile più flessibile, caratteristico di alcuni linguaggi come JavaScript. Questo non si basa su classi che ereditano successivamente l'una dall'altra, ma su catene di prototipi. Posso aggiungere campi agli oggetti, aggiungendoli al corrente layer. Questo consente quindi grande versatilità ma allo stesso tempo non sarà per nulla scontato determinare quale sia il tipo dell'oggetto, data la non esistenza di un effettiva classe.

   Subtyping Strutturale La struttura di un oggetto determina se sia o meno sottotipo di un altro oggetto. Notiamo che si lascia molto spazio al programmatore. Un oggetto B è sottotipo di un oggetto A se contiene almeno tutti i suoi membri pubblici.

2. **Class Based**: Stile più rigido, permette di effettuare controlli statici dato che la classe fornirà all'oggetto un vero e proprio tipo. Questa metodologia è definita **nominal typing**. Si sacrifica della versatilità guadagnando disciplina e possibilità di effettuare controlli statici.

   Subtyping Nominale Se abbiamo definito una classe B come estensione di una classe A allora sappiamo per certo come ricostruire la loro gerarchia. Questo tipo di subtyping implica quello strutturale. Dunque la condizione di subtyping nominale è più forte rispetto a quella strutturale.

Type Weakening Tipo inferito staticamente come "temporaneamente polimorfo", in realtà funziona solo da placeholder a tempo statico per fare in modo che a run time si possa istanziare con un tipo concreto.

4.3 Java

Java è un linguaggio che risale agli anni 90 e cercava di astrarre, deresponsabilizzando i programmatori da problemi noti fino ad allora, come gestione manuale di memoria e puntatori. Questo linguaggio incarna pienamente lo stile object oriented.

4.3.1 Ereditarietà ed Interfacce in Java

Ogni classe in Java può ereditare da una singola classe, risolvendo così tutte le potenziali ambiguità causate da un ereditarietà multipla. Allo stesso tempo però è possibile che una classe implementi più interfacce. Definiamo bene però la differenza tra interfacce e classi:

1. **Tipo Astratto (Interfaccia)**: Fornisce tutti i campi pubblici attesi da un oggetto che implementa quell'interfaccia, ma non stabilisce alcuna implementazione. Questo corrisponde al tipo che verrà controllato a **tempo statico**.
2. **Tipo Concreto (Classe)**: Stabilisce l'implementazione effettiva di ogni campo pubblico atteso. Per questo motivo solitamente le classi implementano delle interfacce. Questo corrisponde al tipo che verrà utilizzato a **tempo dinamico**.

4.3.2 Scoping in Java

Assumendo che l'obiettivo del Java fosse quello di rendere tutto più "static safe", over approssimando ogni potenziale tipo di problema a tempo statico. Questo causa una **non esistenza** dello shadowing di blocco, infatti il Java semplicemente non permette la ridichiarazione di variabili e non permette neanche la dichiarazione senza inizializzazione. Lo scope della variabile dichiarata sarà quindi di blocco.

4.3.3 Modello Memoria JVM

La Java Virtual Machine lavorerà interagendo con una memoria che segue questa classificazione:

1. **Workspace**: Spazio delle classi e delle variabili statiche.
2. **Stack**: Pila dei record di attivazione dei metodi che conterranno le variabili locali.
3. **Heap**: Spazio dove verranno allocate tutte le istanze delle varie classi.

La gestione della memoria verrà eseguita dalla **Garbage Collection** che mostreremo più avanti.

4.3.4 Gerarchia Classi, Overloading/Overriding, Casting, Dispatch Vectors

L'ereditarietà permette di costruire una vera e propria gerarchia tra le varie classi. Tutte le classi saranno sottoclassi di **Object**, ossia la superclasse "originale".

Classe Astratta Una classe astratta è una "via di mezzo" tra un interfaccia ed una classe effettiva. Questo perchè permette di dichiarare dei metodi senza effettivamente fornire una implementazione. Ma non forza questo comportamento, si presenteranno dunque delle classi "a metà", a cui potremo fornire successivamente delle implementazioni esaustive alle segnature fornite.

Classi Annidate Possiamo avere dichiarazioni di classi in classi, e queste possono essere **statiche** se appartengono alla classe stessa, altrimenti appartengono all'istanza.

Overloading Una classe può avere più metodi con lo stesso nome, ma questi per essere distinguibili devono fornire delle segnature diverse.

Overriding Una sottoclasse può "riscrivere" il corpo di un metodo della superclasse. Questi quindi manterranno stesso nome e segnatura ma fornendo implementazioni diverse.

Downcast Una superclasse deve essere castata **esplicitamente** ad un sottotipo, non è un operazione implicita.

Upcast Ogni sottoclasse è già castata **implicitamente** alla sua superclasse, corrisponde ad una superclasse "più specifica".

Dispatch Vectors Possiamo evitare di risalire l'albero delle classi grazie a tutta la staticità fornita dal Java. Infatti, a differenza dello stile *Object Based*, qui abbiamo la possibilità di definire una tabella, detta **Dispatch Vector**, che permette l'accesso ai campi ed ai metodi in tempo costante, anche se li stiamo invocando da un istanza di classe che non li appartiene direttamente.

Questo infatti è il concetto di **condivisione** delle dispatch vectors, le sottoclassi possono fare riferimento a metodi già esistenti oppure creare nuovi riferimenti (aggiungendo righe) non visibili alla superclasse. In caso ad esempio di **overriding** posso cambiare il riferimento della sottoclasse verso la nuova implementazione.

4.3.5 Java Generics, Wildcard e Type Erasure

I Generics sono variabili di tipo, che ci permettono di astrarre dal tipo di una particolare istanza della classe. In questo modo stiamo dando una possibilità ai nostri oggetti di mantenere dati di tipo **generico** ed **uniforme**. Ogni volta infatti questa $< T >$ dovrà essere istanziata con un tipo effettivo. E' come se stessimo **parametrizzando** il tipo che forniremo alla classe.

Wildcard Un tipo particolare di generico. Possiamo definire grazie alle wildcard dei **generici temporanei anonimi**, che verranno istanziati appena gli verrà fornito un tipo effettivo. Definiti solitamente con "?", tornano utili quando il tipo parametrizzato viene utilizzato una sola volta.

Type Erasure Dato che i generici sono stati aggiunti con **Java 5**, per questioni di retrocompatibilità, si è deciso di convertire tutti i tipi parametrici a tipi **object** a tempo di compilazione. Dunque a livello **bytecode** i generici non esistono. Questo causa una perdita del tipo effettivo post-compilazione.

Sottotipi e Generici Ma $list < Integer >$ è sottotipo di $list < Number >$? No, perchè in Java non è valida la covarianza. Se Java ammettesse la covarianza, non garantirebbe il massimo della sicurezza a run time. L'unico complesso in cui vale la covarianza è *Array* per questioni storiche, modificando questa caratteristica si sarebbero creati problemi di retrocompatibilità.

Conclusioni I Generici ci permettono di rafforzare il polimorfismo della soluzione, potendo anche effettuare controlli statici su di essi, ma a tempo di compilazione vengono tutti sostituiti con **Object** cancellando i tipi effettivi istanziati.

4.3.6 Java Collection Framework

Il Java fornisce di default una collezione di strutture dati già implementate che permettono un livello ulteriore di astrazione, dato che ogni volta che avremo la necessita di utilizzare un struttura dati nota non sarà necessario implementarla ex-novo, ma basterà importarla dalla collezione standard.

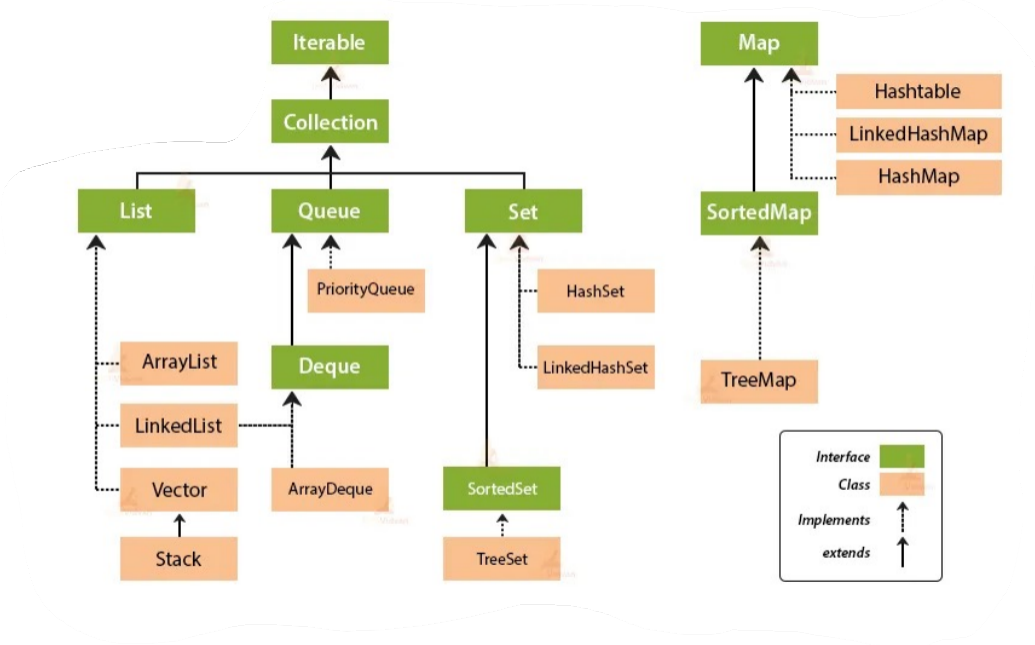


Figure 3: Gerarchia della JCF

Notiamo che i nodi rappresentati in verde rappresentano interfacce, mentre quelle in arancione sono classi. Dunque i nodi arancioni (classi) implementano specifiche interfacce che forniranno correttezza statica

Senza questo framework, le uniche strutture dati disponibili sarebbero state $Vector$ ed $Array$.

Iterator Notiamo che il padre di tutti i nodi della foresta sinistra delle JFC è l'interfaccia iteratore. Questo ci permette di iterare sugli elementi delle collezioni grazie ad un generatore che restituisce un iteratore.

Un generatore solitamente è rappresentato da una funzione che restituisce un iteratore che fornisce specifici metodi per iterare sulla collezione come $itr.hasNext()$ oppure $itr.next()$. Questo ci permette di creare **astrazione** anche sul concetto di iterazioni su collezioni. Sappiamo bene che iterare su elementi di collezioni è una procedura molto ricorrente, grazie a questo espediente riusciamo a categorizzare quindi quel tipo di operazioni ricorrenti.

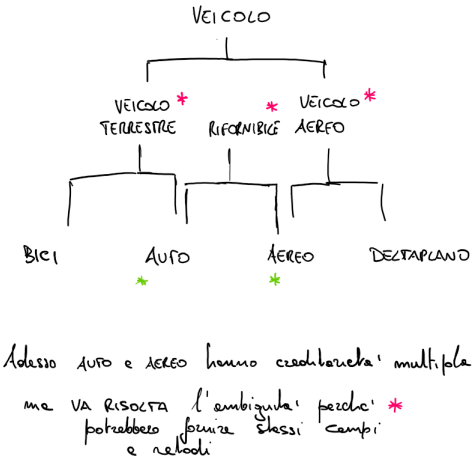
Bisogna avere però l'accortezza di non iterare da due diversi iteratori

sulla stessa collezione. Questo infatti porterebbe ad una situazione di inconsistenza, infatti se un iteratore apportasse una modifica alla collezione, tutti gli altri iteratori verrebbero invalidati.

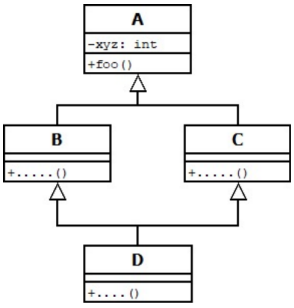
Astrazione e For Each Grazie all'utilizzo dell'iteratore possiamo utilizzare nuovi costrutti più ad alto livello, permettendoci di iterare direttamente sugli elementi di una collezione senza l'utilizzo di effettivi indici, grazie al comando $for(tipo_{elem}\ elem\ :\ collezione)$, che corrisponde al $(for\ \ of)$ in JS.

4.3.7 Ereditarietà Multipla e Mixin

Ricordiamo che Java non supporta l'Ereditarietà Multipla, ma permette ad una classe di implementare più interfacce. In generale altri linguaggi invece permettono l'Ereditarietà Multipla, ma hanno la necessità di gestire la possibilità di metodi denominati ugualmente tra classi ed anche l'occorrenza ripetuta del nodo padre della gerarchia della classe. Mostriamo un paio di esempi:



1. **C++**: Permette ad una classe di ereditare da più superclassi ma questo genera il **diamond problem**, ossia che una potenziale radice super padre della gerarchia potrebbe occorrere più volte durante una risalita dell'albero. C++ risolve il problema permettendo la **virtualizzazione**, dunque conoscendo questa potenziale problematica, questo linguaggio lascia la possibilità allo sviluppatore di "ereditare virtualmente", ossia verranno skippate le superclassi intermedie, l'ipotetico metodo trovato nel padre viene puntato dalla sottoclasse stessa, skippando consapevolmente le superclassi a metà risalita.



2. **Java**: Permette di implementare più interfacce gestendo propriamente le **dispatch vectors**, senza superclassi.

3. **Scala**: Introduce un metodo alternativo, ossia i **Mixin**. Questi sono un **interpretazione** più **dinamica** di una vera e propria **superclasse**. Una classe infatti invece di ereditare da superclassi, si definisce come rimescolamente di altre classi, il mixin infatti può essere rimescolato esso stesso ad una classe.

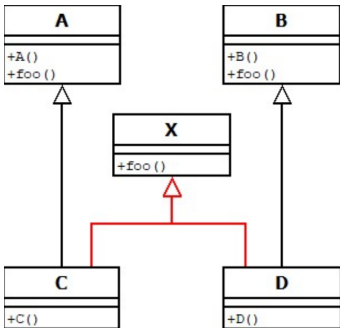
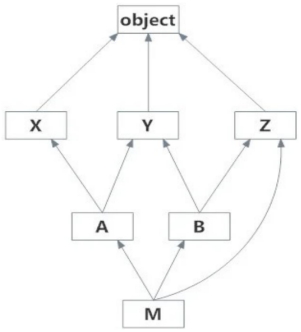


Figure 4: X è un mixin

4. **Python**: Questo linguaggio utilizza invece un altro approccio, cerca infatti di linearizzare la gerarchia delle classi per risolvere l'ambiguita sulla ricorrenza di metodi in classi diverse. Questo avviene grazie ad una sorta di buona enumerazione di tutto l'albero gerarchico, linearizzato da un algoritmo specifico, detto **Method Resolutin Order (MRO)**.



[M, A, X, B, Y, Z, object]

4.3.8 Eccezioni in Java

Suddividiamo le eccezioni (situazioni che interrompono il flusso regolare di esecuzione) in due grandi categorie:

1. **Checked**: Eccezioni che vanno dichiarate, non estendono la classe *Exception*.
2. **Unchecked**: Eccezioni che non vanno dichiarate, ma che vengono lanciate autonomanmente quando occorre un errore a runtime. Queste estendono la classe *Exception*.

4.3.9 Specifiche Metodi e Accenno Semantica Assiomatica

Tutti i metodi sono definiti da una specifica. Questa si compone di:

1. **Precondizione** $\{P\}$: Condizione iniziale su cui viene invocato il metodo. Solitamente indicato da un **REQUIRES**.
2. **Metodo** *met*: Effettiva funzione applicata.
3. **Postcondizione** $\{Q\}$: Condizioni richieste all'uscita dal metodo. Solitamente indicato da un **EFFECTS**.

$$\{P\}\ met\ \{Q\}$$

Questi test permettono di ricavare errori di programmazione persino a tempo statico, fornendo più consistenza.

4.3.10 Principio di Sostituzione di Liskov

Un oggetto di un sottotipo può sostituire un oggetto del supertipo senza influire sul comportamento dei programmi che usano il supertipo.

Stabilire a tempo statico se questo principio viene rispettato è un problema indecidibile, di conseguenza si attuano delle buone pratiche ed il rispetto di determinate proprietà che mantengono il principio valido.

Proprietà da rispettare Elenchiamo le proprietà necessarie al mantenimento di questo principio:

1. **Segnatura**: E' necessario che si preservi la segnatura da supertipo a sottotipo. Se si effettua override è consentito anche ritornare un sottotipo del tipo originale.

2. **Metodi**: I metodi devono mantenere lo stesso comportamento:

   (a) Posso allargare le **precondizioni**.

   (b) Posso restringere le **postcondizioni**.

3. **Proprietà**: Se la superclasse ha delle proprietà, queste devono essere mantenute nella sottoclasse. Nelle slide si mostrava un esempio su una superclasse *IntSet* con proprietà {invarianza, evoluzione}, dunque anche le sottoclassi dovranno rispettare queste proprietà.

   Implicitamente vogliamo che sia sempre rispettato anche l'**incapsulamento**, senza mai permettere l'accesso non permesso a campi privati di una classe.

4.4 Garbage Collection

Dopo aver visto tutti i nuovi livelli di astrazione, possiamo immaginare di volerlo fare anche con la gestione della memoria, dato che la sua gestione manuale viola il concetto base di astrazione della programmazione.

Problematiche Gestione Manuale Memoria Il linguaggi come il C, che lascia la gestione della memoria completamente in mano al programmatore, sono noti due problemi frequenti:

1. **Dangling Pointers**: Dati due puntatori ptr_1 e ptr_2, uno alias dell'altro (puntano alla stessa zona di memoria), se effettuassi una $free(ptr_1)$ allora ptr_2 punterebbe ad una zona non accessibile della memoria a causa dell'aliasing iniziale.
2. **Memory Leak**: Perdere un riferimento ad una zona di memoria prima di deallocarla. Questo genera *garbage*, ossia memoria non riutilizzabile fino a chiusura del programma.

Modello Memoria Analizziamo la composizione della memoria:

1. **Static**: Tabelle di supporto, variabili statiche...
2. **Stack**: Parte di memoria che mantiene le variabili locali dei vari record di attivazione delle funzioni durante il loro ciclo di vita.
3. **Heap**: Memoria detta dinamica, che necessita di essere gestita.

Rootset Solitamente una zona di memoria dell'heap, una volta allocata, viene puntata da delle variabili dello spazio dello stack oppure static. Questa variabile è definita **rootset**.

Tipi di Heap, Frammentazione e Tipi di Fit Un heap può essere **statico** nelle dimensioni oppure **dinamico** se ha la possibilità di ridimensionarsi. Entrambe le tipologie utilizzano una **lista libera** di supporto per tenere traccia delle locazioni occupate e quelle piene.

Un ulteriore problema da gestire nell'heap è la **frammentazione**, ossia spazio utilizzabile suddiviso in tante piccole sezioni. Elenchiamone le varie tipologie:

1. **Interna**: Durante il "deposito" di una x in memoria, occupo uno spazio y, anche se $dim_x << y$.
2. **Esterna**: Tante frazioni di memoria libera ben distribuita tra blocchi di memoria occupata. Sarebbe necessario uno "shift" di fix per ricavare memoria libera unitaria.

4.4.1 Metodi di Raggiungibilità

Elenchiamo tutti i metodi di verifica di raggiungibilità, analizzandone le caratteristiche.

1. **Reference Counting**: Mano mano tengo traccia sulla memoria heap quanti puntatori stanno facendo riferimento a quella zona. Questo funzione, ma solo per il "primo layer" di memoria puntata. Questo infatti funzionerà sui primi riferimenti da rootset ad heap, ma in caso di **puntatori circolari** nell'heap, questi non verranno trovati, causando **memory leak**.

2. **Tracing**: Verifico la raggiungibilità effettuando letteralmente una visita partendo dalla rootset. Le operazioni eseguite dopo la visita stabiliscono delle sottocategorie:

   (a) **Mark & Sweep**: Durante la visita marco su 1 bit tutti i raggiungibili, dunque il resto sarà irragiungibile. Questo mi permette di stabilire che periodicamente dovrò mettere in pausa tutto ed effettuare una pulizia dei non raggiungibili, causando uno "stop the world" event. Questa metodologia ha pro e contro, infatti risolve il memory leak dei riferimenti circolari, ma causa un evento di stop per effettuare la pulizia.

   (b) **Coping Collection**: Descriviamo l'algoritmo di Cheney che caratterizza questa metodologia:

      i. **Divisione Heap**: Ho due sezioni di heap, "from-space" e "to-space".

      ii. **Allocazione**: Uso soltanto la parte "from-space" per allocare nuova memoria.

      iii. **Attivazione GC**: Le celle attive vengono copiate da "from-space" in "to-space".

      iv. **Inverto le parti**: Rendo la "to-space" attiva e la "from-space" disattivata.

      v. **Lista Libera**: Restituisco in una lista libera tutta la parte liberata.

3. **Generational Garbage Collector**: Gestione dell'heap in base allo scope dei riferimenti, quindi chi viene creato di recente probabilmente sarà in un blocco più interno che presto si chiuderà. Questo causa la suddivisione dell'heap in generazioni, dove quelle **young** vengono cancellate e quelle **old** mantenute.

4. **Hotspot JVM**: Java suddivide tutto l'heap in 3 generazioni, nella prima e nella seconda applica il Copying Collection, mentre nella terza applica Mark & Sweep cercando di evitare la frammentazione.

4.4.2 Piccolo confronto con Rust

L'approccio seguito da Rust è completamente differente, infatti non fa uso di garbage collectors ma impone delle regole sulla **ownership** di ogni oggetto sull'heap. Questo permette una gestione della memoria safe ed alternativa. Elenchiamone delle caratteristiche:

(a) **Ownership**: Ogni zona di memoria ha un sono **owner** (puntatore **rootset**), durante un assegnamento di puntatori viene **scambiata** la **ownership**. Quando si esce dallo scope dell'**owner**, la memoria sull'heap viene deallocata.

(b) **Borrowing**: Esiste il **borrow** in lettura o in scrittura, dunque l'**aliasing** si crea esplicitamente nello scope dell'**owner**. L'obiettivo è quello di non far coesistere aliasing e mutation, dunque segue delle regole:

i. Il riferimento è **unico** e **mutabile**.

ii. Il riferimento **non è unico** ed è **immutabile**.

5 Concorrenza

Il primo concetto fondamentale è capire come un calcolatore moderno riesca a darci

l'impressione che tutti i **thread** (sottoprocessi) vengano eseguiti contemporaneamente. Per assurdo se volessimo eseguire realmente k processi in parallelo, allora sarebbe necessario avere anche k CPU.

Sistemi Multicore Questa tipologia di sistemi possiede effettivamente multipli **core**, ma in ogni caso i k processi saranno comunque di più rispetto agli n core.

$$k >> n$$

Concorrenza e Interleaving Dunque la situazione reale è descritta da un **mescolamento** di esecuzioni di **thread**. Questo fenomeno è detto **interleaving**.

Non determinismo e Scheduler Non è possibile descrivere l'ordine di esecuzione di thread con un automa deterministico, perchè l'ordine verrà settato da uno **scheduler**, che ha il ruolo di non far collidere due esecuzioni di processi, gestendo l'**interleaving** sopra definito. Dobbiamo anche considerare che una CPU non esegue un intero comando ad alto livello, ma ha la necessita di dividerlo in tante **istruzioni ASM**, dunque già una sola istruzione corrisponde a tante azioni diverse eseguite dalla CPU.

Condivisione di memoria tra processi Due diversi processi non condivideranno mai la stessa memoria senza un preciso protocollo di sincronizzazione. Solo successivamente definiremo il **memory sharing**, ma prima di quello specifico modello si assume che due macro processi non condividano mai memoria.

5.1 Shared Memory vs Message Passing

Mostriamo le differenze tra questi due principali modelli di sincronizzazione di processi (in questo caso thread).

1. **Shared Memory**: Si accede alle stesse zone di memoria, adattando il comportamento dei due thread in base al valore contenuto nella zona di memoria. Mostriamo l'esempio **produttore/consumatore**:

```
1    // variabili globali (memoria condivisa)
2    int x = 0;
3    // THREAD 1
4    producer(){
5        int k = 6;
6        x = fattoriale(k);
7    }
8    // THREAD 2
9    consumer(){
10       while(x==0) {sleep(10);}
11       print(x);
12   }
```

   Notiamo come il secondo thread, grazie all'approccio di **busy waiting**, sta assumendo un comportamento rispetto al valore di quella zona di memoria. Questo approccio è molto inefficiente dato che il secondo thread impegna la CPU solo per un continuo check della variabile.

2. **Message Passing**: Questo modello invece sincronizza le esecuzioni dei due thread inviando "messaggi di check". Mostriamo un esempio in pseudocodice:

```
1    // variabili globali (memoria condivisa)
2    int x = 0;
3    // THREAD 1
4    producer(){
5        int k = 6;
6        x = fattoriale(k);
7        send(x);
8    }
9    // THREAD 2
10   consumer(){
11       int y = receive();
12       print(y);
13   }
```

   In questo modo il **consumer** si mette in attesa del **producer** e il *send* permetterà lo sblocco del **consumer** con il valore atteso.

5.1.1 Processi vs Thread

Definiamo differenze tra thread e processi.

1. **Processi**: Elenchiamone le caratteristiche:
   (a) Ogni processo ha uno **spazio di memoria indipendente**. Si garantisce maggiore sicurezza ed isolamento.
   (b) I processi **comunicano** tramite **meccanismi** forniti dal **sistema operativo**.
   (c) I processi sono più **"pesanti"** in termini di risorse, dato che il **sistema operativo allocherà risorse** indipendenti per questi ultimi.
   (d) Supportano il reale **parallelismo**.
   (e) Il **crash** di un processo non influenza l'esecuzione degli altri, grazie all'isolamento definito prima.
2. **Thread**: Elenchiamone le caratteristiche:
   (a) Possono **condividere** tra di loro **memoria**.
   (b) La **comunicazione** tra thread **avviene** proprio grazie alla **memoria** che condividono.
   (c) In termini di **risorse** i **thread** sono **meno "pesanti"**.
   (d) Anche i thread supportano il parallelismo spaziale su core diversi.
   (e) Il **crash** di questi uno di questi può compromettere anche altri thread, dato che la memoria su cui operano è condivisa.

5.2 Mini Linguaggio Imperativo ed Estensione Concorrente

Immaginiamo un semplice linguaggio[1] imperativo definito dalle correnti regole su grammatica e semantica, e successivamente estendiamolo con costrutti di **concorrenza**.

Sintassi:

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e; e$$

Regole di tipo

Γ mi dice solo se la locazione esiste

$$\Gamma \vdash n : \text{int} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Gamma(\ell) = \text{intloc}}{\Gamma \vdash !\ell : \text{int}}$$

$$\frac{\Gamma(\ell) = \text{intloc} \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash \ell := e : \text{unit}} \qquad \frac{\Gamma \vdash e_1 : \text{unit} \quad \Gamma \vdash e_2 : \text{unit}}{\Gamma \vdash e_1; e_2 : \text{unit}}$$

$$\Gamma \vdash skip : \text{unit}$$

Semantica (small-step)

Le **configurazioni** sono coppie $\langle e, s \rangle$ dove e è un'**espressione** ed s è uno **stato** che associa locazioni a interi

$$\text{(op+)} \quad \langle n_1 + n_2, s \rangle \to \langle n, s \rangle \quad \text{se } n = n_1 + n_2$$

$$\text{(comp1)} \quad \frac{\langle e_1, s \rangle \to \langle e_1', s' \rangle}{\langle e_1 + e_2, s \rangle \to \langle e_1' + e_2, s' \rangle} \qquad \text{(comp2)} \quad \frac{\langle e_2, s \rangle \to \langle e_2', s' \rangle}{\langle n + e_2, s \rangle \to \langle n + e_2', s' \rangle}$$

$$\text{(deref)} \quad \langle !\,\ell, s \rangle \to \langle n, s \rangle \quad \text{se } \ell \in dom(s) \text{ e } s(\ell) = n$$

Aggiornamento della memoria

$$\text{(assign1)} \quad \langle \ell := n\,, s \rangle \to \langle skip, s + \{\ell \mapsto n\} \rangle \quad \text{se } \ell \in dom(s)$$

In generale, la valutazione di un'espressione può portare a modificare lo stato

$$\text{(assign2)} \quad \frac{\langle e, s \rangle \to \langle e', s' \rangle}{\langle \ell := e, s \rangle \to \langle \ell := e', s' \rangle}$$

$$\text{(seq1)} \quad \langle skip\,; e_2\,, s \rangle \to \langle e_2, s \rangle \qquad \text{(seq2)} \quad \frac{\langle e_1, s \rangle \to \langle e_1', s' \rangle}{\langle e_1; e_2\,, s \rangle \to \langle e_1'; e_2\,, s' \rangle}$$

[1] Acquisizioni dirette delle slide del corso di Paradigmi di Programmazione 24/25

5.2.1 Estensione Concorrente

Aggiungiamo questi costrutti e regole d'inferenza per rendere il linguaggio capace di eseguire **composizioni di espressioni**.

Sintassi estesa:

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e;e \mid e|e$$

Regole semantiche aggiunte:

$$\text{(parallel1)} \quad \frac{\langle e_1, s\rangle \to \langle e_1', s'\rangle}{\langle e_1 | e_2, s\rangle \to \langle e_1' | e_2, s'\rangle}$$

$$\text{(parallel2)} \quad \frac{\langle e_2, s\rangle \to \langle e_2', s'\rangle}{\langle e_1 | e_2, s\rangle \to \langle e_1 | e_2', s'\rangle}$$

Notiamo che nella grammatica abbiamo fornito **operazioni atomiche**, ossia:

1. ! dereferenziazione
2. := assegnamento
3. *skip*

5.2.2 Lock - Meccanismo di Mutua Esclusione (Mutex)

Forniamo a questo linguaggio un modo per poter limitare l'accesso a specifiche risorse solo ad un thread alla volta.

Lock/Unlock I *lock* sono un esempio di primitiva di sincronizzazione. Stiamo assicurando che per una determinata quantità di tempo **un solo** thread avrà accesso a specifiche risorse. Successivamente forniamo anche un **unlock** per poter sbloccare le risorse bloccate. Dunque l'accesso a delle zone di memoria richiede specifiche fasi:

1. Acquisizione di un mutex (token), con il costrutto lock.
2. Accesso esclusivo, se nessun altro thread sta già accedendo a quella memoria si può procedere con le operazioni del thread.
3. Attesa, se un altro thread ha già bloccato la zona di memoria con un mutex, allora il corrente thread è messo in attesa.
4. Rilascio, quando il thread ha finito, rilascia la memoria effettuando un unlock del rispettivo token.

Sintassi estesa:

$$e ::= n \mid e + e \mid !\ell \mid \ell := e \mid skip \mid e;e \mid e|e \mid \textbf{lock}\, m \mid \textbf{unlock}\, m$$

dove $m \in \mathbb{M}$ (insieme, anche infinito, dei mutex)

Regole semantiche aggiunte:

Solo se il mutex è libero

$$\text{(lock)} \quad \langle \textbf{lock}\, m,\ s, M\rangle \to \langle (),\ s,\ M + \{m \mapsto \textbf{true}\}\rangle \ \text{if}\ \neg M(m)$$

$$\text{(unlock)} \quad \langle \textbf{unlock}\, m,\ s, M\rangle \to \langle (),\ s,\ M + \{m \mapsto \textbf{false}\}\rangle$$

5.3 Tipologie di Lock e Deadlocks

L'utilizzo coerente e consistente dei lock permette una corretta gestione degli accessi concorrenti alla memoria, non permettendo a più thread di operare sulla stessa memoria contemporaneamente. Dunque ogni locazione condivisa dovrebbe essere associata ad un mutex e ogni volta che un thread inizia ad operare su quella zona di memoria dovrebbe bloccarla e rilasciarla quando ha terminato.

5.3.1 Coarse/Fine Grained Locking

Mostriamo due esempi "polarizzati" di utilizzo di lock:

1. **Coarse Grained**: Questa strategia utilizza un unico mutex per proteggere tutte le locazioni. Questo porta ad essere sicuri di non causare mai un **deadlock**, ma **riduce** il **livello** di **concorrenza** generabile.

2. **Fine Grained**: Questa strategia utilizza un mutex per ogni locazione bloccata dai thread. In questo modo il **livello** di **concorrenza** generabile è **alto** ma si può andare in contro a **deadlock**.

5.3.2 Deadlocks e Risoluzione

Un **deadlock** è una situazione nella quale l'utilizzo improprio dei lock causa un blocco dell'intero programma. L'esempio classico è quello dell'**attesa circolare**.

$$\begin{gathered}\text{(ATTESA CIRCOLARE)} \\ Thread_1 \ blocca \ Risorsa_1 \ ed \ attende \ Risorsa_2 \\ mentre \\ Thread_2 \ blocca \ Risorsa_2 \ ed \ attende \ Risorsa_1\end{gathered} \tag{1}$$

Questa è una delle cause di blocco dei programmi che vanno killati. Esistono però metodologie per prevenire, evitare o risolvere queste occorrenze del deadlock:

1. **Deadlock Prevention**: Si stabiliscono precise regole **statiche** tali per cui non possono verificarsi deadlock. Mostriamo esempio di regola:

   (a) **2-Phase Locking**: Si stabilisce un ordine su tutti i mutex, dunque ogni thread dovrà:

      i. Eseguire dei lock in ordine **crescente**.

      ii. Dopo aver operato sulla memoria, esegue degli unlock in ordine **decrescente**.

2. **Deadlock Avoidance**: Si effettuano controlli a **runtime** che permettono di stabilire se si sta per causare un deadlock e si evita di incorrere in quella situazione.

3. **Deadlock Recovery**: Si permette al programma di cadere in un deadlock, ma se ciò accade a **runtime** si ripristina uno stato senza deadlock.

5.4 Concorrenza nei Linguaggi Moderni

Analizziamo dei tipi di concorrenza in linguaggi moderni.

5.4.1 Java e Multithreading

Java offre la classe *Thread* che permette la gestione della concorrenza. Mostriamo un esempio:

```
1   class Main {
2       public static void main(String[] args) {
3           Runnable r = new Runnable() {
4           // Creazione classe anonima
5           public void run() { // Sovrascrittura
6               System.out.println("Hello Thread!");
7               }
8           };
9           Thread t = new Thread(r);
10          // Creazione del thread
11          t.start(); // Avvio del thread
12      }
13  }
```

Dunque questo schema fornisce uno stack per ogni oggetto *Thread*, ma la gestione dell'Heap è in *Shared Memory*.

Locking in Java Anche il Java permette di il lock di risorse utilizzate da un thread.

1. **Synchronized [Metodo] - Coarse Grained**: Il costrutto **synchronized** permette di lockare le risorse utilizzate da un thread. Dunque verranno bloccate tutte le risorse utilizzate fino alla conclusione di un determinato metodo.
2. **Synchronized [Blocco] - Fine Grained**: Quando non vogliamo bloccare tutte le risorse di un metodo possiamo utilizzare il *synchronized* di blocco, ridimensionando la portata del lock. Questo utilizzo può causare deadlocks.

5.4.2 Esempi in altri linguaggi

1. **JavaScript**: Offre *callback*, *Promise*, *async/await* che regolano la sincronizzazione in maniera "funzionale".
2. **Go/Erlang**: Permettono la sincronizzazione (message passing) tramite canali di comunicazione.
3. **Kotlin**: Gestisce la concorrenza tramite un albero di gerarchia delle attività.
4. **Haskell**: Gestisce le transazioni di database, ossia operazioni atomiche eseguite su memoria condivisa: in caso di conflitto vengono annullate e ripetute.