Programmazione II

Luca De Paulis

13 aprile 2021

Indice

Parte I

OBJECT-ORIENTED PROGRAMMING

1

OOP in Java

1.1 CLASSI E OGGETTI IN JAVA

Java, come tutti i linguaggi Object-Oriented, si basa fortemente sulle nozioni di *classe* e di *oggetto*.

- Un oggetto è un insieme strutturato di *variabili di istanza*, che rappresentano lo stato interno dell'oggetto, e di *metodi*, che rappresentano le azioni che possiamo compiere sull'oggetto.
- Una classe, invece, è un *template* che possiamo usare per creare oggetti di un determinato tipo: la definizione di una classe specifica
  - tipo e valori iniziali dello stato locale degli oggetti;
  - l'insieme delle operazioni che possono essere eseguite su oggetti che sono istanze di quella classe.

  In particolare ogni definizione di classe implementa uno o più metodi costruttore: essi servono a "costruire" un oggetto di quella determinata classe.

Per implementare il concetto di *information hiding* gli oggetti in Java nascondono all'esterno il loro stato locale, ma hanno un'interfaccia ben definita, data dai loro metodi pubblici, che consentono di agire sull'oggetto solo nei modi concessi dal programmatore.
Gli oggetti sono caratterizzati da:

- uno stato interno
- un nome che individua ogni oggetto
- un ciclo di vita (creazione, riferimento, disattivazione)
- una locazione di memoria
- dei comportamenti, dati dai metodi.

Il meccanismo utilizzato per gli assegnamenti tra oggetti è il cosiddetto *sharing strutturale*: l'assegnamento `obj1 = obj2` (dove `obj1` e `obj2` sono due oggetti) fa in modo che `obj1` sia un *riferimento* all'oggetto riferito da `obj2`. Non viene quindi creata una copia di `obj2`, ma ora due variabili si riferiscono alla stessa locazione di memoria: questo concetto va sotto il nome di *aliasing*.

1.1.1 Definizione di classe

Vediamo ora come si definisce una classe in Java.

```
public class Point {
    private int x, y;

    public Point(int x0, int y0){
        x = x0;
        y = y0;
    }

    public void add(Point p){
        x += p.x;
        y += p.y;
    }

    public Point sum(Point p){
        Point res = new Point(x + p.x, y + p.y);
        return res;
    }

    public String toString(){
        return "("+ x + ", " + y + ")";
    }
}
```

- **`public`** e **`private`** sono dei *modificatori*: **`public`** si usa per dire che un metodo, una variabile di istanza o la dichiarazione di una classe devono essere visibili al codice esterno, mentre **`private`** si usa per nascondere la dichiarazione al resto del codice.

  Generalmente le variabili di istanza devono essere nascoste (per rispettare il principio dell'information hiding), mentre i metodi sono pubblici in modo da consentire lo scambio di dati tra le istanze di una classe e l'esterno.

- La parola chiave **`class`**, seguita da un nome, è l'inizio della dichiarazione della classe.

- Nella prima parte si dichiarano le *variabili di istanza* dela classe: in questo caso `x` e `y` sono dichiarate private in quanto vogliamo che siano visibili e modificabili solo dalla classe.

- Il primo metodo dichiarato è il *costruttore della classe*: esso ha lo stesso nome della classe e ci consente di creare nuovi oggetti di tipo `Point` inizializzando le variabili di istanza.

- Seguono poi dei *metodi*: essi permettono al resto del codice di operare con oggetti di tipo `Point`, ad esempio sommandoli tra loro o stampandoli a schermo tramite il metodo `toString`.

Un programma Java è mandato in esecuzione invocando un metodo speciale, detto *main*, con la seguente firma:

```
public static void main(String[] args)
```

Un esempio potrebbe essere il seguente:

```
public class Main {
public static void main(String args[]){
    Point a = new Point(1, 0);
    Point b = new Point(0, 2);
    Point c = a.sum(b);
```

```
        System.out.println(a.toString() + " + " + b + " = " + c);
    }
}
```

Il metodo `main` crea due oggetti di tipo `Point` tramite la parola chiave **`new`**: essa manda in esecuzione il costruttore della classe, creando due oggetti che rappresentano rispettivamente il punto $(1, 0)$ e il punto $(1, 2)$. Successivamente viene invocato il metodo `sum` dell'oggetto `a`: questo metodo restituisce un nuovo oggetto di tipo `Point` che contiene il punto dato dalla "somma" dei punti `a` e `b`. Infine il metodo stampa i tre punti sfruttando il metodo `toString`.

1.2 ABSTRACT STACK MACHINE

Per descrivere il funzionamento di Java sfrutteremo un modello computazionale chiamato *Abstract Stack Machine*: essa ci consente di descrivere i cambiamenti dello stato e le interazioni tra gli oggetti.

La ASM è formata da tre componenti:

- un *workspace*, dove sono memorizzati i programmi in esecuzione e quindi le istruzioni da eseguire;
- uno *stack*, che viene usato per gestire i binding tra variabili e locazioni di memoria che contengono gli oggetti;
- uno *heap* che contiene le locazioni di memoria degli oggetti e viene usato per la gestione della memoria dinamica.

Supponiamo ad esempio di avere una dichiarazione di classe di questo tipo:

```
public class Node{
    private int elt;
    private Node next;

    public Node(int e0, Node n0){
        elt = e0;
        next = n0;
    }
    ...
}
```

Se creiamo un oggetto di tipo `Node`:

```
Node first = new Node(5, null);
```

la ASM si modificherà nel seguente modo:

- nello stack comparirà un nuovo binding: il nome di variabile `first` sarà un riferimento ad una locazione di memoria sullo heap, in cui sarà contenuto un nuovo oggetto di tipo `Node`;
- nello heap verrà allocato lo spazio per il nuovo oggetto: in particolare vi sarà lo spazio per le variabili di istanza (inizializzate a 5 e a **`null`** rispettivamente) e dello spazio aggiuntivo che analizzeremo in seguito.

È importante rendersi conto che i riferimenti di Java sono simili ai puntatori definiti in C/C++, ma non consentono l'uso dell'aritmetica dei puntatori: sono semplicemente *riferimenti* agli oggetti allocati sullo heap.

1.3 INTERFACCE

Abbiamo visto che un meccanismo per definire nuovi tipi in Java è il meccanismo delle classi. Tuttavia Java definisce anche altri costrutti sintattici per realizzare nuovi tipi di dati, come ad esempio le *interfacce*.

Un'interfaccia serve a definire il tipo di un oggetto in modo dichiarativo: vengono dichiarati i metodi che gli oggetti di quel tipo devono implementare, ma non viene definita la loro implementazione.

Ad esempio, consideriamo gli oggetti che vengono rappresentati graficamente su uno schermo bidimensionale: ognuno di essi ha una posizione rispetto all'asse delle ascisse e delle ordinate; inoltre ognuno di essi può essere spostato sullo schermo muovendo le sue coordinate in un altro punto dello schermo. Per rappresentare ciò in Java dichiariamo un'interfaccia:

```
public interface Displaceable{
    public int getX();
    public int getY();
    public void mode(int dx, int dy);
}
```

Questa interfaccia dichiara tre metodi:

- il metodo `getX`, che (quando implementato) dovrà restituire la posizione sulle ascisse del nostro oggetto;
- il metodo `getY`, che (quando implementato) dovrà restituire la posizione sulle ordinate del nostro oggetto;
- il metodo `mode`, che prende in input due interi (`dx` e `dy`) e avrà lo scopo di spostare l'oggetto dalla posizione corrente, aggiungendo `dx` alle ascisse e `dy` alle ordinate.

Le interfacce sono utili in quanto possono essere *implementate dalle classi*:

```
public class Point implements Displaceable {
    private int x, y;

    public Point(int x0, int y0){
        x = x0;
        y = y0;
    }

    public int getX(){
        return x;
    }

    public int getY(){
        return y;
    }

    public void move(int dx, int dy){
        x += dx;
        y += dy;
    }
}
```

La classe `Point` implementa i metodi definiti dall'interfaccia `Displaceable`:

- la parola chiave **`implements`** serve a specificare che la classe sta implementando un'interfaccia;

- ogni classe può implementare più di un interfaccia;
- quando una classe implementa un'interfaccia *deve* fornire un'implementazione di ogni metodo definito nell'interfaccia.

Ovviamente una classe può implementare più metodi di quelli definiti da un'interfaccia, come si può vedere nel prossimo esempio:

```
public class ColorPoint implements Displaceable {
    private Point p;
    private Color c;

    public ColorPoint(int x0, int y0, Color c0){
        p = new Point(x0, y0);
        c = c0;
    }

    public int getX(){
        return p.getX();
    }

    public int getY(){
        return p.getY();
    }

    public Color getColor(){
        return c;
    }

    public void move(int dx, int dy){
        p.move(dx, dy);
    }
}
```

La classe `ColorPoint` ha più metodi di quelli definiti nell'interfaccia `Displaceable`; inoltre notiamo che essa delega il compito di restituire la posizione sulle ascisse/ordinate e il compito di spostare il punto alla classe `Point`: ciò è una buona pratica di programmazione in quanto consente il riuso del codice. Infatti se volessimo estendere le funzionalità di `mode` in questo caso possiamo semplicemente modificare l'implementazione contenuta nella classe `Point`: la classe `ColorPoint` sarebbe automaticamente aggiornata e non dovremmo cambiare anche il suo codice.

1.3.1 Interfacce come tipi di variabili

Possiamo usare le interfacce per dichiarare nuove variabili:

```
Displaceable d;
d = new Point(3, 1);
d.move(1, -1);
d = new ColorPoint(0, 0, new Color("white"));
```

La variabile `d` è di tipo `Displaceable`: possiamo assegnare ad essa una qualsiasi istanza di una classe che implementa `Displaceable`, come ad esempio `Point` oppure `ColorPoint`.

Questo fenomeno è chiamato *sub-typing*: la variabile `d` è di tipo `Displaceable`, dunque può essere legata ad un qualsiasi oggetto che implementi l'interfaccia `Displaceable`. Più in generale, un tipo `A` è detto *sottotipo* di un altro tipo `B` se `A` soddisfa tutti gli obblighi richiesti da `B`.

1.3.2 Interfacce multiple

Come abbiamo detto prima, un oggetto può implementare più di un'interfaccia. Ad esempio dichiariamo questa nuova interfaccia:

```
public interface Area{
    public void getArea();
}
```

Una classe che implementa sia `Displaceable` che `Area` può essere la seguente:

```
public class Circle implements Area, Displaceable{
    private Point center;
    private int rad;

    public Circle(int x0, int y0, int r0){
        center = new Point(x0, y0);
        rad = r0;
    }

    public double getArea(){
        return Math.pi * rad * rad;
    }

    public int getX(){
        return center.getX();
    }

    public int getY(){
        return center.getY();
    }

    public void move(int dx, int dy){
        center.move(dx, dy);
    }
}
```

1.4 EREDITARIETÀ

Consideriamo il seguente codice:

```
public class PaintedCircle {
    private Point center;
    private int radius;
    private Color fillColor;
    private Color borderColor;
    private double borderThickness;

    public void fillWith(Color c){
        fillColor = c;
    }

    public void setBorderThickness(double t){
        borderThickness = t;
    }

    public void setBorderColor(Color c){
```

```
            borderColor = c;
        }
        ...
    }

    public class PaintedTriangle {
        private Point v1, v2, v3;
        private Color fillColor;
        private Color borderColor;
        private double borderThickness;

        public void fillWith(Color c){
            fillColor = c;
        }

        public void setBorderThickness(double t){
            borderThickness = t;
        }

        public void setBorderColor(Color c){
            borderColor = c;
        }
        ...
    }
```

Il codice delle due classi è corretto e funziona, ma non è buon codice: se volessimo modificare il funzionamento dei bordi, ad esempio aggiungendo la possibilità di avere bordi tratteggiati, dovremmo modificarlo per ognuna delle due classi `PaintedCircle` e `PaintedTriangle`. Per questo si ricorre al meccanismo dell'*ereditarietà tra classi*.

L'ereditarietà ci consente di

- riusare il codice e creare una gerarchia tra le classi, in modo che
  - le classi in alto nella gerarchia rappresentino tipi più generalizzati/astratti;
  - le classi in basso nella gerarchia rappresentino tipi più specifici/concreti;
- sfruttare il meccanismo del sub-typing per usare tipi più specifici dove sono previsti i loro supertipi.

Il secondo punto viene generalmente chiamato *principio di sostituzione*: se `B` è un sottotipo di `A`, allora gli oggetti di tipo `B` possono essere usati dove sono previsti oggetti di tipo `A`. In particolare

- un'istanza del sottotipo (`B`) soddisfa sempre le proprietà del supertipo (`A`);
- un'istanza del sottotipo può avere più vincoli del supertipo.

1.4.1 Ereditarietà in Java

L'ereditarietà tra due classi `B` e `A` viene realizzata in Java tramite la parola chiave **`extends`**:

```
class B extends A {...}
```

Al contrario delle interfacce, una classe in Java può estendere una sola classe: questo meccanismo è chiamato *ereditarietà singola* (al contrario dell'ereditarietà multipla, come ad esempio accade in C++).

Facciamo un esempio di ereditarietà:

```
public class D {
    private int x, y;

    public int addBoth(){
        return x + y;
    }
}

public class C extends D {
    private int z;

    public int addThree(){
        return addBoth() + z;
    }
}
```

Notiamo che:

- la classe `C` eredita tutte le variabili di istanza di `D` implicitamente; tuttavia, dato che le variabili di istanza di `D` sono dichiarate private, gli oggetti di tipo `C` non possono accedervi (anche se sono presenti!);
- la classe `C` eredita tutti i metodi di `D` e quindi (se sono dichiarati **`public`**) può usarli all'interno dei suoi metodi.

Modificatore `protected`

Per fare in modo che le variabili di istanza siano nascoste all'esterno ma visibili alle sottoclassi il Java mette a disposizione la parola chiave **`protected`**: se dichiarassimo protette (invece che private) le variabili della classe `D` riusciremmo ad accedervi e a modificarle dalla classe `C`.

La scelta **`private`**/**`protected`** dipende dalla situazione e non vi sono regole generali per capire quando è meglio usare l'una oppure l'altra.

Il metodo `super`

Il metodo costruttore della classe padre non viene ereditato direttamente: per chiamarlo è necessario usare il metodo **`super`**, come mostrato dal seguente esempio:

```
public class D {
    private int x, y;

    public D(int initX, int initY){
        x = initX;
        y = initY;
    }
    ...
}

public class C extends D {
    private int z;

    public C(int initX, int initY, int initZ){
        super(initX, initY);
        z = initZ;
    }
}
```

This

La parola chiave `this` ci permette di riferirci all'oggetto corrente. Anche se solitamente possiamo usare i metodi e le variabili dell'oggetto corrente senza farli precedere dal nome dell'oggetto e dall'operatore punto, in alcuni casi può essere utile.

Uno di questi casi è quando vogliamo disambiguare i nomi delle variabili di istanza con i nomi delle variabili in ingresso al metodo:

```
public class Point {
    private int x, y;
    public Point(int x, int y){
        this.x = x;
        this.y = y;
    }
}
```

Nel metodo costruttore `x` e `y` si riferiscono ai parametri formali del metodo, mentre `this.x` e `this.y` si riferiscono alle variabili di istanza dell'oggetto.

Un altro uso per `this` è come costruttore implicito:

```
public class Rectangle{
    private int x, y;
    private int width, height;

    public Rectangle(int x0, int y0, int w, int h){
        x = x0;
        y = y0;
        width = w;
        height = h;
    }

    public Rectangle(int w, int h){
        this(0, 0, w, h);
    }

    public Rectangle(){
        this(0, 0, 0, 0);
    }
}
```

In questo caso la classe `Rectangle` ha tre diversi costruttori: il primo prende 4 parametri (posizione sull'asse x, sull'asse y, larghezza e altezza), il secondo ne prende solo 2 (larghezza e altezza) e infine il terzo non prende parametri. Il costruttore implicito `this` usa il codice del costruttore con 4 parametri per definire i costruttori con 2 e 0 parametri, ponendo a 0 i termini non specificati.

Upcasting e downcasting

L'ereditarietà in Java ci consente di usare oggetti sottotipo al posto di oggetti supertipo, in modo da avere codice più modulare. Possiamo inoltre trasformare un oggetto di un supertipo in un oggetto di un sottotipo e viceversa, tramite l'*upcasting* e il *downcasting*.

L'upcasting serve a trasformare una variabile di un sottotipo in una variabile di un supertipo:

```
public class Vehicle {...}
public class Car extends Vehicle {...}

...
Vehicle v = (Vehicle) new Car();
```

Invece il downcasting serve per trasformare una variabile di un supertipo e specializzarla in un sottotipo:

```
Car c = (Car) new Vehicle();
```

L'upcasting può essere anche implicito, mentre il downcasting va sempre esplicitato e può avvenire solo tra una classe e una sua super-classe.

Ogni classe estende `Object`

Esiste una classe in Java che fa da "antenato" a tutte le altre classi: la classe `Object`. Infatti tutte le classi estendono `Object` implicitamente, poiché in `Object` sono definiti alcuni metodi che devono essere comuni a tutti gli oggetti.

Metodo `equals`

Il metodo `equals` viene usato per controllare la *deep equality* tra due oggetti: si usa `equals` quando si vuole verificare non se due variabili sono riferimenti allo stesso oggetto (*shallow equality*), ma quando si vuole verificare che due oggetti (che occupano due posizioni in memoria diverse) hanno lo stesso *stato interno*.

Siccome il concetto di *deep equality* dipende dal tipo di oggetto che stiamo considerando, il metodo `equals` va ridefinito per ogni nuova classe: la definizione di default controlla solo la *shallow equality* tra due oggetti (tramite l'operatore ==).

Metodo `toString`

Il metodo `toString` permette di rappresentare lo stato interno di un oggetto sotto forma di stringa. Anche in questo caso il metodo va ridefinito per ogni classe.

Metodo `clone`

Il metodo `clone` permette di creare un nuovo oggetto con lo stesso stato interno dell'oggetto corrente (anche se in alcuni casi può creare una situazione di condivisione di dati). Questo metodo viene ereditato solo se la classe implementa l'interfaccia `Cloneable`.

1.5 TIPI DINAMICI E STATICI

Consideriamo il seguente esempio:

```
public class Vehicle {...}
public class Car extends Vehicle {...}

...
Vehicle v = (Vehicle) new Car();
```

Qual è il tipo di `v`?

- Da una parte `v` è una variabile di tipo `Vehicle`.
- D'altro canto l'oggetto il cui riferimento è contenuto in `v` è di tipo `Car`, un sottotipo di `Vehicle`.

Definiamo quindi due *tipi* associati ad ogni variabile:

- Il *tipo statico*, che è dato da tutte le informazioni di carattere sintattico (in particolare, dal tipo della variabile). Nel nostro caso il tipo statico di `v` è `Vehicle`.
- Il *tipo dinamico* che rappresenta il tipo dell'oggetto istanziato sullo heap riferito dalla variabile in questione. Nel nostro caso, il tipo dinamico è `Car`.

Facciamo un esempio:

```
public class Father {
    ...
    public void printA(){
        System.out.println("A");
    }
    public void printB(){
        System.out.println("B");
    }
}
public class Son extends Father {
    ...
    public void printB(){
        System.out.println("B from Son");
    }
    public void printC(){
        System.out.println("C");
    }
}

...
Father obj = new Son();
obj.printA();
obj.printB();
obj.printC();
```

Il metodo `printA()` stampa "A" a schermo, come dovrebbe. I metodi successivi si comportano in maniera più particolare:

- Il metodo `printB()` stampa "B from Son": siccome il tipo dinamico della variabile `obj` è `Son`, dunque il metodo `printB()` usa l'implementazione fornita dalla classe `Son` e non dalla classe padre.
- Il metodo `printC()` dà un errore di compilazione: siccome il tipo statico della variabile `obj` è `Father` il compilatore non riconosce il metodo `printC()` e quindi dà errore, anche se il tipo dinamico lo consentirebbe.

Il motivo di questo comportamento è il seguente: siccome una variabile di tipo `Father` può contenere un riferimento ad oggetti di un qualunque suo sottotipo, a priori gli unici metodi che siamo certi che essa può eseguire sono quelli definiti da `Father`. Infatti tutti i metodi della classe padre devono essere ereditati o ridefiniti dalle classi che ereditano da essa, mentre i metodi definiti dai sottotipi non sono sempre accessibili (come nel caso di `printC()`).

Invece il punto dei sottotipi è quello di specializzare le classi supertipo, dunque se un metodo della classe supertipo viene ridefinito, la versione invocata sarà sempre la più specializzata possibile (come nel caso di `printB()`).

1.6 ECCEZIONI IN JAVA

In alcuni casi vogliamo segnalare il fatto che un metodo non può restituire il risultato corretto perché si è verificato un errore (ad esempio, si vuole leggere un file che non esiste). Il Java fornisce un potente strumento sintattico per gestire questi casi: le cosiddette *eccezioni*.

Le eccezioni sono particolari oggetti che servono a segnalare una situazione anomala: esse sono uno strumento più potente di scegliere "valori particolari" da restituire in caso di errore in quanto

- possiamo avere eccezioni specializzate per ogni tipo di errore;

- c'è *separations of concerns*: abbiamo un costrutto sintattico che si occupa di gestire gli errori.

Per sollevare un'eccezione bisogna usare la parola chiave **`throw`**:

```
throw new MyException();
```

Esso richiede come argomento un qualunque oggetto che sia un sottotipo della classe `Throwable`. Questa classe ha una famiglia di sottoclassi molto ramificata, su cui torneremo più avanti per una distinzione particolare.

Gestione delle eccezioni

Per gestire le eccezioni il Java mette a disposizione il costrutto **`try - catch - finally`**.

- Nella clausola **`try`** si inseriscono le istruzioni che potrebbero sollevare un'eccezione. Ad esempio

  ```
  try {
      int[] arr = new int[3];
      System.out.println(arr[4].toString);
  }
  ```

  Siccome l'array ha solo 3 elementi, tentando di accedere alla posizione di indice 4 viene sollevata una `IndexOutOfBoundsException`, che sarà catturata da un eventuale blocco **`catch`**.

- Nelle clausole **`catch`** si specifica un'eccezione che può essere sollevata dal codice contenuto nella **`try`**: nel caso questa eccezione venisse sollevata, il codice all'interno della specifica clausola **`catch`** viene eseguito. Ad esempio:

  ```
  try {
      int[] arr = new int[3];
      System.out.println(arr[4].toString);
  } catch (ArrayOutOfBoundsException e) {
      System.out.println(e);
  } catch (NullPointerException e) {
      System.out.println(e);
  }
  ```

  In questo caso, dopo che `IndexOutOfBoundsException` viene sollevata, essa viene catturata dal primo blocco **`catch`** che quindi esegue il suo codice e fa proseguire l'esecuzione del programma alla fine del blocco **`try - catch`**. Se l'eccezione sollevata non compare in nessuno dei blocchi **`catch`**, l'esecuzione del metodo si interrompe con un fallimento e l'eccezione viene passata al metodo chiamante.

- Il blocco **`finally`** viene eseguito alla fine dell'esecuzione **`try - catch`**, in *qualsiasi* caso. Infatti anche se nessuno dei blocchi **`catch`** riesce a catturare l'eccezione sollevata nel blocco **`try`**, il metodo esegue il blocco **`finally`** prima di restituire il controllo al chiamante. La clauolsa **`finally`** è utile per eseguire del codice di *clean-up* a prescindere dal fatto che sia stata sollevata un'eccezione o meno.

Osserviamo che il blocco **`catch`** non può catturare solamente l'eccezione che viene specificata come argomento, ma anche tutti i suoi sottotipi: ad esempio in

```
try {
    int[] arr = new int[3];
    System.out.println(arr[4].toString);
} catch (Throwable e) {
    System.out.println(e);
}
```

la `catch` cattura qualsiasi tipo di eccezione, in quanto tutte sono sottotipo di `throwable`.

Inoltre possiamo avere più blocchi `try - catch` annidati.

```
try {
    try {
        x = Array.searchSorted(v, y);
    } catch (NullPointerException e) {
        throw new NotFoundException();
    }
} catch (NotFoundException e) {
    System.out.println(e);
}
```

In questo caso il blocco `catch` esterno può catturare sia la `NotFoundException` sollevata dal blocco `catch` interno, sia una possibile `NotFoundException` sollevata dal metodo `Array.searchSorted`.

1.6.1 Eccezioni checked e unchecked

Come abbiamo detto prima la classe `Throwable` ha diverse sottoclassi. Esse si dividono in due categorie: le classi che generano eccezioni *checked* e quelle che generano eccezioni *unchecked*.

ECCEZIONI *CHECKED*

Le eccezioni *checked* sono tutte le eccezioni che sono sottotipo della classe `Exception`. Esse sono delle eccezioni particolari in quanto devono essere elencate nelle firme dei metodi che possono sollevarle, come in

```
public void myMethod() throws MyCheckedException { ... }
```

Inoltre le eccezioni checked non possono essere propagate, ma devono essere gestite (tramite `try - catch`) appena vengono sollevate (a meno che il metodo corrente non le dichiari nella sua firma).

Queste condizioni vengono controllate dal compilatore, per cui le eccezioni checked sono le eccezioni che non "distruggono il flusso del programma": basta catturare l'eccezione e si può continuare con l'esecuzione.

ECCEZIONI *UNCHECKED*

Un'eccezione *unchecked* è un'eccezione che estende `RuntimeException`: esse rappresentano tutti gli errori che possono accadere a runtime, come ad esempio la possibilità di avere un riferimento a `null` (codificato dalla `NullPointerException`), oppure di essere andati fuori dagli indici permessi in un array (errore codificato dalla `IndexOutOfBoundsException`), o tante altre.

Al contrario delle eccezioni checked, le unchecked non devono necessariamente essere enumerate nella firma di un metodo (anche se è buona pratica farlo ugualmente), né devono essere obbligatoriamente catturate da una specifica clausola `try - catch`.

Le eccezioni checked sono più sicure e robuste delle unchecked, poiché il compilatore si assicura che vengano elencate e gestite; tuttavia quando siamo ragionevolmente sicuri che l'eccezione non verrà sollevata può essere più utile usare un'eccezione unchecked. Al contempo,

le eccezioni unchecked possono essere difficili da notare, in quanto non vengono dichiarate esplicitamente e il programmatore non sa quale metodo le ha sollevate: in questo caso l'unica cosa da fare è separare il blocco `try` in più blocchi, in modo da sapere effettivamente quale metodo solleva la specifica eccezione.

1.6.2 Definire nuove eccezioni

Per definire una nuova eccezione bisogna innanzitutto decidere se la si vuole definire checked o unchecked. Nel primo caso, essa deve essere un sottotipo di `Exception`, e deve contenere al suo intero solamente uno o più costruttori:

```
public MyCheckedException extends Exception {
    public MyCheckedException(){
        super();
    }

    public MyCheckedException(String e){
        super(e);
    }
}
```

Il discorso è identico per eccezioni unchecked, con la differenza che la classe deve estendere `RuntimeException`:

```
public MyUnCheckedException extends RuntimeException {
    public MyUnCheckedException(){
        super();
    }

    public MyUnCheckedException(String e){
        super(e);
    }
}
```

1.6.3 Introduzione alla Programmazione Difensiva

Lo stile di programmazione che usa le eccezioni (insieme, come vedremo nel prossimo capitolo, ai contratti d'uso) per evitare situazioni anomale viene chiamato *defensive programming* o programmazione difensiva. In questo stile di programmazione il programmatore descrive quali sono gli input ammessi per ogni metodo definito, e in caso di input non ammessi solleva un'eccezione per segnalare all'utente che il metodo non può svolgere il suo compito correttamente. Parleremo più approfonditamente di defensive programming più avanti.

2

ADT e Specifiche

2.1 DESIGN-BY-CONTRACT

Il concetto di *design-by-contract* è stato introdotto per permettere di costruire progetti robusti e facilmente estendibili. Secondo questo modello di progettazione è necessario creare una *barriera di astrazione* tra colui che progetta il software e il cliente tramite un *contratto d'uso*.

Il contratto è formato da due parti:

- una *precondizione* (clausola *requires*), che serve a colui che progetta il software per dire quali devono essere i vincoli sui dati prima della chiamata di un metodo;
- una *postcondizione* (clausola *effects*), che indica l'effetto del metodo nei casi in cui la precondizione è soddisfatta: il metodo deve sottostare precisamente alla postcondizione, ad esempio modificando le giuste variabili, oppure sollevando le eccezioni specificate, eccetera.

Facciamo un esempio:

```
// @requires: num >= 0
// @effects: returns the square root of the number num
public static double sqrt(double num);
```

In questo caso la precondizione è che il parametro in ingresso `num` sia non-negativo (se ciò non fosse l'operazione di radice quadrata non avrebbe senso); la postcondizione invece è che il metodo (se la precondizione è verificata) restituisce effettivamente la radice quadrata del numero.

Il contratto d'uso serve ad astrarre l'uso di un metodo dalla sua implementazione: non serve conoscere il codice sorgente di un metodo per poterlo usare. Nel caso del metodo `sqrt` non importa sapere *in che modo* esso calcola la radice quadrata: il contratto d'uso ci dice che il parametro in ingresso deve essere non-negativo e ci assicura che in questo caso il risultato sia davvero la radice quadrata del numero, a prescindere da quale sia l'algoritmo usato per calcolarla.

Questo concetto si rivela molto utile quando si vogliono fare modifiche incrementali ai metodi: se volessimo migliorare l'implementazione di un metodo non dobbiamo preoccuparci di come esso viene usato, ma dobbiamo assicurarci solamente di mantenere inalterate la precondizione e la postcondizione. In questo modo il cliente non vede alcuna differenza nell'uso del metodo precedente con quello nuovo e non possono esserci errori dovuti all'implementazione di nuove funzionalità, in quanto tutte le richieste e tutte le funzionalità vanno inserite nelle clausole *requires* e *effects*.

2.2 DEFENSIVE PROGRAMMING

Le precondizioni e le postcondizioni viste nella sezione precedente possono essere pensate come formule logiche della Logica del Primo Ordine. La relazione tra le due (ovvero che se vale la precondizione allora dobbiamo assicurarci che, alla fine del metodo, valga la postcondizione) può essere realizzata tramite una semplice implicazione:

$$\text{Pre} \implies \text{Post}.$$

Tuttavia un'implicazione è sempre vera quando l'antecedente (in questo caso `Pre`) è falso, dunque il contratto ci permette di fare qualsiasi cosa quando l'antecedente è falso. Ad esempio il seguente programma rispetta il contratto d'uso dato:

```
// @requires: num ≥ 0
// @effects: returns the 4th root of the number num
public static double fourthRoot(double num) {
    if(num < 0) {       // anything's allowed
        return 254;
    } else {
        return sqrt(sqrt(num));
    }
}
```

Infatti, siccome abbiamo specificato chiaramente che il programma funziona correttamente solamente quando `num` è non-negativo, ci aspettiamo di non trovarci mai nel primo caso, per cui potremmo direttamente eliminare il costrutto **`if`**.

Tuttavia una buona pratica di programmazione è controllare due volte la precondizione: la prima tramite la clausola requires, la seconda attraverso l'uso delle eccezioni.

```
// @requires: num ≥ 0
// @effects: returns the 4th root of the number num
public static double fourthRoot(double num)
            throws IllegalArgumentException {
    if(num < 0) {
        throw new IllegalArgumentException();
    } else {
        return sqrt(sqrt(num));
    }
}
```

In questo caso siamo certi che il programma non può comportarsi in modi diversi da come lo vogliamo poiché, oltre ad aver esplicitato le precondizioni, abbiamo anche sollevato un'eccezione adatta in caso il metodo fosse stato chiamato con i parametri errati.

Questo stile di programmazione si chiama *programmazione difensiva* (oppure *defensive programming*), e serve a creare programmi facili da usare, facili da mantenere e al contempo sicuri (poiché blocchiamo qualsiasi possibile input che non rispetta le precondizioni tramite le eccezioni).

2.3 ABSTRACT DATA TYPES

Abbiamo visto che il concetto di specifica ci permette di astrarre l'uso di un metodo dalla sua implementazione: possiamo fare la stessa cosa con le strutture dati, che sono la base di ogni progetto. Infatti può capitare spesso che una scelta di struttura dati fatta ad inizio progetto possa rivelarsi non ottimale in seguito a causa della sua implementazione: vorremmo quindi un modo per astrarre dall'organizzazione e dal significato specifico dei dati e pensare solo in termini delle operazioni fornite.

Un esempio di tipo di dato astratto può essere una classe in Java da un punto di vista esterno: per operare sulla classe possiamo solamente usare i metodi che ci vengono forniti dalla classe, insieme alle loro specifiche; se essi in futuro dovessero essere migliorati (ad esempio dal punto di vista della performance) il cliente non avrà modo di rendersene conto.

I metodi di un ADT devono essere nascosti all'utente se non per la loro specifica: ognuno di essi dovrà avere delle precondizioni e delle postcondizioni che permettono a chi usa il metodo di sfruttarlo anche senza conoscerne l'implementazione. I metodi di un ADT possono essere divisi in 4 categorie a seconda del loro scopo:

- i *creators*, che servono a creare nuove istanze dell'ADT (in Java sono i costruttori);
- i *producers*, che sono metodi che restituiscono nuovi dati;
- i *mutators*, che modificano i dati esistenti;
- gli *observers*, che servono a dare informazioni relative allo stato interno degli oggetti, facendo attenzione a non violare la barriera di rappresentazione.

Infine gli ADT possono essere divisi in due gruppi: quelli *mutable*, che permettono operazioni che ne modifichino lo stato interno, e quelli *immutable*, che non hanno *mutators*.

Facciamo due esempi.

2.3.1 ADT Poly

Clausole Overview e Typical Object

Supponiamo di voler creare un ADT che rappresenti un polinomio. Il primo passo è specificare *cosa rappresenta* il nostro ADT e qual è un elemento tipico di questo tipo di dato astratto.

```
/**
 * A Poly is an immutable polynomial with integer coefficient.
 * A typical element is
 *          c_0 + c_1 x + c_2 x^2 + ...
 */
public class Poly { ...
```

La clausola *overview* stabilisce se il tipo di dato astratto è mutable o immutable e ne definisce il modello astratto tramite il *typical element*.

Creators

I metodi *creators* servono a creare una nuova istanza del tipo di dato astratto.

```
/**
 * @effects: makes a new Poly = 0
 */
public Poly();

/**
 * @throws: NegExponentException if n < 0
 * @effects: makes a new Poly = cx^n
 */
public Poly(int c, int n);
```

Siccome i creators servono a creare nuovi oggetti, l'unica clausola indispensabile è la clausola effects.

OBSERVERS

I metodi *observers* servono a reperire parte dell'informazione contenuta nello stato interno di un'istanza di un ADT.

```
/**
 * @returns: the degree of this,
 *      the greatest exponent with a
 *      non-zero coefficient.
 *      Returns 0 if this = 0.
 */
public int degree() { }

/**
 * @throws: NegExponentException if n < 0
 * @returns: the coefficient of the term
 *      of this whose exponent is n.
 */
public int coeff(int n) { }
```

È importante che gli observers non modifichino lo stato astratto, né violino la barriera di astrazione esponendo dati all'esterno, ad esempio restituendo il riferimento ad un oggetto contenuto nello stato interno. Inoltre notiamo che nella descrizione degli observers si usa sempre la rappresentazione astratta dell'oggetto fornita tramite la clausola overview.

PRODUCERS

I *producers* servono a produrre nuovi oggetti del tipo dell'ADT a partire da oggetti già esistenti. Essi non devono avere effetti laterali, ovvero non devono modificare lo stato astratto degli oggetti su cui sono chiamati, ma devono solamente produrre nuovi dati.

```
/**
 * @returns: this + p
 */
public Poly add(Poly p) { }

/**
 * @returns: this * p
 */
public Poly mult(Poly p) { }
```

2.4 IMPLEMENTARE ADT

Abbiamo visto come la specifica e i concetti del design-by-contract ci consentano progettare tipi di dati robusti. Per implementare un tipo di dato astratto abbiamo bisogno di altri due importanti strumenti: l'*invariante di rappresentazione* e la *funzione di astrazione*.

INVARIANTE DI RAPPRESENTAZIONE

L'invariante di rappresentazione (abbreviato ad RI, per *rapresentation invariant*) è una funzione che prende un oggetto e ritorna un valore di verità (vero o falso): esso serve per stabilire se l'istanza considerata è ben formata, ovvero rispetta delle condizioni particolari. L'invariante di rappresentazione è una guida per chi implementa, modifica o verifica il funzionamento del tipo di dato astratto: nessun oggetto deve violare l'invariante di rappresentazione.

Ad esempio, se volessimo codificare una struttura dati che rappresenta un insieme matematico, potremmo richiedere che non vi siano mai due valori uguali nell'insieme: l'invariante di rappresentazione conterrà questa clausola e quindi le istanze ben formate della classe saranno soltanto quelle che non contengono duplicati.

Funzione di astrazione

La funzione di astrazione (o AF, dall'inglese *abstraction function*) è una funzione che serve ad interpretare astrattamente un'istanza del tipo di dato astratto: essa è definita solo sugli oggetti che rispettano l'invariante di rappresentazione e ci permette di pensare agli oggetti come se fossero la loro rappresentazione astratta.

Ad esempio, nel caso degli insiemi la funzione di astrazione prende un oggetto della classe `Set` e restituisce l'insieme $\{ a_1, a_2, \ldots, a_n \}$ che contiene tutti i dati dell'oggetto originale.

La funzione di astrazione ci consente di pensare alle varie operazioni eseguibili sulla classe `Set` come se fossero effettivamente insiemi matematici.

Facciamo un esempio.

```
public interface CharSet{
    // Overview: CharSet is a finite and
    // modifiable set of characters.
    // A typical element is
    //          {c1,...,cn}.

    //@effects: creates a new and empty CharSet
    public CharSet() {...}

    //@modifies: this
    //@effects: this_pre = this_post ∪ {c}
    public void insert(Character c) {...}

    //@modifies: this
    //@effects: this_pre = this_post \ {c}
    public void delete(Character c) {...}

    //@effects: returns true if and only if c is an element of this
    public void member(Character c) {...}

    //@effects: return cardinality of this
    public int size() {...}
}
```

Consideriamo ora la seguente implementazione:

```
    public class ArrayListCharSet{
        private List<Character> elems = new ArrayList<Character>();

        public void insert(Character c) { elems.add(c); }
        public void delete(Character c) {
            int i = elems.indexOf(c);
            if (i > -1) elems.remove(i);
        }
        public void member(Character c) {
            return elems.contains(c);
        }
        public int size() {
            return elems.size();
        }
    }
```

Parte II

LINGUAGGI DI PROGRAMMAZIONE

3

Sintassi di OCaml

3.1 VALORI ED ESPRESSIONI

OCaml è un linguaggio multiparadigma derivato dal linguaggio funzionale CaML e dai suoi antenati nella famiglia di ML. Essendo alla base un linguaggio funzionale, un programma OCaml è un'*espressione* che può essere valutata ad un *valore*, ovvero ad un'espressione che non deve essere valutata ulteriormente.

Useremo la notazione <exp> $\Longrightarrow$ v oppure la notazione Eval(exp) = v per dire che l'espressione exp viene valutata al valore v.

Il primo metodo per creare espressioni è il costrutto `let`. Ad esempio l'espressione

```
let x = 42;;
```

ci permette di *legare* al nome della variabile x il valore 42. Questo costrutto ci consente tuttavia di definire espressioni più complesse:

```
let x = 7*3
    in x*x;;
```

Questa espressione lega alla variabile x il valore 21, cioè il valore a cui l'espressione 7*3 viene valutata; inoltre, questo legame è *locale* all'espressione che segue la parola chiave `in`. Il valore dell'espressione è quindi 441.

3.2 COSTRUTTO `let`

Il costrutto `let` ci permette inoltre di introdurre una nozione di *scope*: nel seguente codice la variabile x è definita e visibile dentro il `let` più esterno, mentre la variabile y esiste ed è visibile solo all'interno del secondo `let`.

```
let x = 42
    in x + (let y = "3110"
                in int_of_string y);;
```

Il valore di questa espressione è 3152, in quanto il valore dell'espressione `let` interna è il numero intero 3110, mentre il valore dell'espressione più esterna è dato dalla valutazione di x + 3110, che restituisce 3152.

Regola di valutazione del costrutto `let`

Studiamo ora la regola formale di valutazione del **`let`**:

$$\frac{\text{Eval(e1)} = v' \quad \text{subst(e2, x, } v') = \text{e2}' \quad \text{Eval(e2}') = v}{\text{Eval(}\texttt{let x = e1 in e2}\text{)} = v}.$$

Questa regola ci dice che la valutazione dell'espressione **`let`** `x = e1` **`in`** `e2` è v se:

- la valutazione di `e1` è un valore v';
- sostituendo il valore v' al posto di tutte le occorrenze libere di `x` nell'espressione `e2` otteniamo l'espressione `e2'`;
- la valutazione di `e2'` è esattamente il valore v.

Esempio 3.2.1. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 2 + (5 * 3) in x + 10;;
```

(1) Valutiamo l'espressione `2 + (5 * 3)`, il cui valore è 17.
(2) Sostituiamo 17 al posto di ogni occorrenza libera di `x` nell'espressione `x + 10`, ottenendo l'espressione `17 + 10`.
(3) Valutiamo quest'ultima espressione, ottenendo 27, che è il valore dell'espressione iniziale.

Esempio 3.2.2. Usiamo la regola di inferenza per calcolare il valore dell'espressione

```
let x = 12 - 3
    in let y = 2 + x
           in x == y;;
```

(1) Valutiamo l'espressione `12 - 3`, il cui valore è 9.
(2) Sostituiamo 9 al posto di ogni occorrenza libera di `x` nell'espressione **`let`** `y = 2 + x` **`in`** `x == y`, ottenendo l'espressione **`let`** `y = 2 + 9` **`in`** `9 == y`.
(3) Valutiamo quest'ultima espressione: siccome anch'essa è un costrutto **`let`** bisogna applicare nuovamente la regola di inferenza:
    (i) Valutiamo l'espressione `2 + 9`, il cui valore è 11.
    (ii) Sostituiamo 11 al posto di ogni occorrenza libera di `y` nell'espressione `9 == y`, ottenendo l'espressione `9 == 11`.
    (iii) Valutiamo l'ultima espressione: siccome $9 \neq 11$ il risultato dell'espressione è `false`.

Segue quindi che il codice iniziale ha valore `false`.

3.3 FUNZIONI

Come in ogni linguaggio funzionale in OCaml le funzioni sono elementi del primo ordine: sono dunque espressioni, esattamente come ogni costrutto del linguaggio.

Sintassi

La sintassi per dichiarare una funzione è la seguente:

```
let <fun_name> (<par_1> : <type_1>) ... (<par_n> : type_n) : ret_type =
    <fun_body>.
```

Per fare un esempio:

```
let f (x : int) : int =
    let y = x * 10
        in y * y;;
```

Una funzione è quindi formata da

- un nome di funzione, in questo caso `f`;
- una lista di parametri, in questo caso il singolo parametro `x`. Notiamo che abbiamo esplicitato il tipo di `x` tramite la notazione `(x : int)`, anche se può essere sottointeso;
- un tipo di ritorno, in questo caso `int`, anch'esso opzionale;
- un corpo di funzione, in questo caso dato dal `let` interno.

Inferenza di tipi

L'inferenza dei tipi dell'interprete OCaml ci permette di dichiarare funzioni senza esplicitare i tipi degli argomenti e/o del risultato, come nel seguente caso:

```
let f x = x + 3;;
```

Siccome l'operatore di somma è specifico per il tipo `int` sia l'argomento della funzione che il suo risultato dovranno essere di tipo intero: il tipo di questa funzione sarà quindi denotato con `int -> int`.

Chiamata di funzione

Per chiamare una funzione non è necessario usare le parentesi: se dichiarassi la funzione `plus` in questo modo

```
let plus x y = x + y;;
```

l'espressione `plus 2 3` sarebbe valutata a `5`.

Possiamo anche dichiarare delle funzioni interne al corpo di un'espressione `let`:

```
let square x = x*x
    in square 3 + square 4;;
```

Questa espressione ha valore `25` poiché l'interprete calcola i valori di `square 3` e `square 4` utilizzando la formula data dal `let`, ovvero `let square x = x * x`.

Funzione ricorsiva

Per dichiarare funzioni ricorsive bisogna usare la parola chiave `rec`. Ad esempio la funzione fattoriale può essere implementata in questo modo:

```
let rec fact x =
    if x = 0
        then 1
        else x * fact (x-1);;
```

Ancora una volta il meccanismo di inferenza dei tipi assegna alla funzione `fact` il tipo `int -> int`.

CURRYING

Consideriamo nuovamente la funzione `plus` definita sopra. Se chiedessimo all'interprete OCaml di ricavarne il tipo, la risposta sarebbe che la funzione `plus` ha tipo `int -> int -> int`, ovvero la funzione prende un intero e restituisce un'altra funzione `int -> int`. Possiamo quindi sfruttare questo fatto per ottenere una funzione applicata parzialmente:

```
let incr = plus 1;;
```

Il tipo della funzione `incr` è `int -> int` (ovvero prende un intero e restituisce un intero), e semplicemente questa funzione si comporta come una `plus` con il primo parametro fissato ad `1`.

FUNZIONI ANONIME

Possiamo dichiarare funzioni con una sintassi diversa usando la parola chiave **`fun`** oppure **`function`**.

```
let f =
    function x y = x * y;;
```

La funzione `f` è semplicemente una funzione che prende due parametri interi (chiamati `x` e `y`) e ne restituisce il prodotto, dunque è equivalente a

```
let f x y = x * y;;
```

Regola di valutazione di una funzione

La regola formale di valutazione di una chiamata di funzione è la seguente:

$$\frac{\text{Eval}(\texttt{f}) = \texttt{fun x\_1 ... x\_n = e} \quad (\forall i : \text{Eval}(\texttt{e\_i}) = v_i) \quad \text{subst}(\texttt{e}, \texttt{x\_1}, \ldots, \texttt{x\_n}, v_1, \ldots, v_n) = \texttt{e}' \quad \text{Eval}(\texttt{e}') = v}{\text{Eval}(\texttt{f e\_1 ... e\_n}) = v}.$$

La regola ci dice quindi che la chiamata di `f e_1 ... e_n` viene valutata ad un valore v se:

- `f` viene valutata ad una funzione (**`fun`**) che prende n parametri (`x_1 ... x_n`) e restituisce un'espressione che dipende da questi parametri (`e`);
- i parametri attuali `e_1 ... e_n` vengono valutati a dei valori $v_1, \ldots, v_n$;
- sostituendo all'interno dell'espressione `e` ogni parametro formale `x_i` con il suo valore v_i si ottiene una nuova espressione `e'`;
- la valutazione di `e'` dà come risultato v.

Esempio 3.3.1. Consideriamo la funzione `plus` definita sopra e la chiamata `plus 5 4`. Allora:

- `plus` viene valutata ad una funzione che prende due parametri e ne restituisce la somma, ovvero:

$$\text{Eval}(\texttt{plus}) = \texttt{fun x y = x + y}.$$

- I parametri vengono valutati rispettivamente a 5 e a 4.
- Sostituendo i valori 5 e 4 nell'espressione `x + y` (al posto di `x` e `y` rispettivamente) otteniamo l'espressione `5 + 4`.
- Valutando quest'ultima espressione otteniamo 9, cioè il valore restituito dalla funzione.

Funzioni di ordine superiore

Siccome le funzioni in OCaml sono oggetti del primo ordine possiamo usarle come parametri di altre funzioni, come in questo esempio:

```
let compose (f : int -> int) (g : int -> int) (x : int) : int =
    g(f x);;
```

La funzione `compose` prende tre parametri:

- una *funzione* `f` da interi in interi;
- una *funzione* `g` da interi in interi;
- un valore intero `x`.

Il risultato della funzione `compose` è la composizione matematica delle funzioni `f` e `g`.

Un altro possibile esempio è il seguente:

```
let rec n_times (f : int -> int) (n : int) : (int -> int) =
    if n = 0 then (fun x -> x)
             else compose f (n_times f (n-1));;
```

Questa funzione restituisce la composizione di una funzione `f` con se stessa per `n` volte usando la ricorsione.

Polimorfismo

Se chiedessimo all'interprete di inferire il tipo della funzione `compose` definita sopra, rimuovendo le annotazioni di tipo, la risposta sarebbe:

```
compose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

Le variabili `'a`, `'b` e `'c` sono *variabili di tipo*: ciò significa che la funzione `compose` non accetta solo parametri di tipo `int`, ma accetta tipi qualsiasi fintanto che tutti i tipi legati ad `'a` siano uguali, tutti i tipi legati a `'b` siano uguali e stessa cosa anche per `'c`.

3.4 LISTE

Una lista in OCaml è una collezione di valori dello stesso tipo. Le seguenti sono tutte liste:

```
let l1 = [1; 2; 3];;
let l2 = 1 :: 2 :: 3 :: [];;
let l3 = [];;
let l4 = 'a' :: 'b' :: ['c', 'd'];;
```

Il simbolo `[]` rappresenta una lista vuota, pertanto la lista `l3` ha tipo polimorfo `'a list`, mentre le altre liste hanno tipo `int list` oppure `char list`.

L'operatore `::` (chiamato *cons*) ci permette di aggiungere un elemento in testa ad una lista (a patto che il tipo dell'elemento sia uguale al tipo della lista). In realtà la sintassi con le parentesi quadre (ad esempio nel caso della prima lista) è del tutto equivalente alla sintassi con l'operatore cons (come nel caso della seconda lista), dunque `l1 = l2`.

Pattern matching

Per scrivere comodamente funzioni su liste è possibile usare il *pattern matching*:

```
let empty lst =
    match lst with
    | []    -> true
    | x::xs -> false;;
```

L'espressione `match lst with` ci permette di confrontare il parametro `lst` con alcuni "pattern", in modo da poter dare la risposta a seconda del pattern a cui la lista viene associata. Quindi se `lst` viene associata alla lista vuota `[]` significa che `lst` è vuota, dunque la funzione `empty` restituisce `true`; invece se `lst` viene associata ad una lista della forma `x::xs` significa che `lst` contiene almeno un elemento (cioè `x`), dunque non è vuota e pertanto restituiamo `false`.

Osserviamo che

- il pattern `[]` viene associato soltanto ad una lista vuota;
- il pattern `x::xs` viene associato ad una lista con *almeno* un elemento (`xs` rappresenta il resto della lista, che può essere vuoto oppure contenere altri elementi);
- il pattern `x::[]` viene associato ad una lista con *esattamente* un elemento;
- il pattern `x::y::xs` viene associato ad una lista con *almeno* due elementi;
- il pattern `x::y::[]` viene associato ad una lista con *esattamente* due elementi;

e così via.

Option types

Alcune funzioni su liste non possono essere applicate ad ogni tipo di lista, ma solo a liste non vuote. Un esempio è la funzione che calcola il massimo di una lista:

```
let rec max_list lst =
    match lst with
    | []    -> ???
    | [x]   -> x
    | x::xs -> max x (max_list xs);;
```

Per risolvere questi problemi (senza ricorrere all'uso del `null`) OCaml implementa gli *option types*:

```
let rec max_list lst =
    match lst with
    | [] -> None
    | x::xs -> match (max_list xs) with
                | None   -> x
                | Some v -> Some (max x v);;
```

Studiamo più nel dettaglio il funzionamento di `max_list`:

- se `lst` è `[]` allora il risultato è un valore particolare, chiamato `None` per indicare l'assenza di valore;
- se `lst` ha un elemento in testa `x` e una coda `xs`, allora la funzione
  - calcola ricorsivamente il valore della coda e lo usa come parametro del pattern matching;

- se il risultato di `max_list xs` è `None`, allora la lista `xs` è vuota e il valore massimo della lista `x::xs` è `x`;
- altrimenti se il valore massimo della lista `xs` è `Some v`, allora il valore massimo della lista `x::xs` è il massimo tra `x` e `v`.

Notiamo che se il risultato non è `None` allora deve essere racchiuso dal costruttore di tipo `Some`: infatti la funzione deve restituire un valore di un tipo preciso e non può restituire talvolta option types e talvolta tipi standard.

Il tipo di questa funzione è quindi `max_list : 'a list -> 'a option`, dove l'`option` indica il fatto che possiamo restituire `None`.

3.5 NUOVI TIPI DI DATO

OCaml ci permette di dichiarare nuovi tipi di dato tramite la parola chiave `type`:

```
type day =
    | Monday
    | Tuesday
    | Wednesday
    | Thursday
    | Friday
    | Saturday
    | Sunday;;
```

Questo costrutto dichiara un nuovo tipo (`day`) che può assumere i valori `Monday`, `Tuesday`, ..., `Sunday`.

Per dichiarare funzioni che operano su valori di tipo `day` possiamo usare il pattern matching:

```
let string_of_day d =
    match d with
    | Monday    -> "Monday"
    | Tuesday   -> "Tuesday"
    | Wednesday -> "Wednesday"
    | Thursday  -> "Thursday"
    | Friday    -> "Friday"
    | Saturday  -> "Saturday"
    | Sunday    -> "Sunday";;
```

I nuovi tipi di dato possono anche "trasportare valori" di tipi standard:

```
type foo =
    | Nothing
    | Int of int
    | Pair of int * int
    | String of string;;
```

Dunque i seguenti sono tutti valori di tipo `foo`:

```
Nothing;;
Int 3;;
Pair (2, 7);;
String "ciao";;
```

Il sistema di creazione di tipi di OCaml è molto potente in quanto ci consente di creare tipi ricorsivi:

```
type bool_expr =
    | True
    | False
    | And of bool_expr * bool_expr
    | Or of bool_expr * bool_expr
    | Not of bool_expr;;
```

Il tipo `bool_expr` è quindi un tipo usato per rappresentare espressioni booleane con gli operatori And, Or e Not: ad esempio i seguenti valori sono di tipo `bool_expr`.

```
True;;
And (False, True);;
Or (Not (And (True, True)), Or (Not True, False));;
```

Scriviamo ora una funzione per valutare un'espressione booleana di tipo `bool_expr`:

```
let rec bool_eval expr =
    match expr with
    | True         -> true
    | False        -> false
    | Or (e1, e2)  -> bool_eval e1 || bool_eval e2
    | And (e1, e2) -> bool_eval e1 && bool_eval e2
    | Not e1       -> not (bool_eval e1);;
```

4

Sintassi e semantica

4.1 SINTASSI

Per descrivere un linguaggio di programmazione abbiamo bisogno di diversi strumenti:

- una **grammatica libera dal contesto** per descrivere la sintassi;
- un metodo per descrivere le regole di scoping e il sistema dei tipi, chiamato **semantica statica**;
- un metodo per descrivere i comportamenti del linguaggio, detto **semantica dinamica**.

Per analizzare la sintassi di un linguaggio abbiamo bisogno quindi di studiarne la grammatica: tuttavia questo può essere scomodo in alcune circostanze, in quanto possono presentarsi problemi di ambiguità tra espressioni.

Si ricorre quindi all'uso dei cosiddetti **Alberi di Sintassi Astratta** (abbreviati in AST, dall'inglese Abstract Syntax Tree): ogni nodo dell'albero rappresenta un'operazione e i suoi nodi figli rappresentano gli operandi dell'operazione (e possono essere a loro volta altre operazioni).

4.2 MECCANISMI DI INFERENZA

Per asserire proprietà (statiche o dinamiche) dei nostri programmi abbiamo bisogno di un meccanismo di inferenza.

Definizione 4.2.1 **judgment.** Si dice **judgment** (o anche **sentenza**) un enunciato che asserisca una proprietà di un oggetto.

Ad esempio, la frase "Il numero 3 è dispari" è un judgment.

Definizione 4.2.2 **Regola di inferenza.** Siano $J_1, \dots, J_n$, J delle sentenze. Una **regola di inferenza** è un'implicazione della forma

$$\frac{J_1 \quad \dots \quad J_n}{J},$$

il cui significato è che se $J_1, \dots, J_n$ sono sentenze derivabili dal sistema assiomatico, allora lo è anche J.

I judgment $J_1, \dots, J_n$ vengono detti *premesse* o *precondizioni*, mentre il judgment J viene detto *conclusione* o *postcondizione*.

Una regola di inferenza senza premesse si dice *assioma*.

Ad esempio la seguente regola è un assioma:

$$\frac{}{0 : \text{nat},}$$

mentre la seguente è una regola con premesse:

$$\frac{n : \text{nat}}{s(n) : \text{nat},}$$

dove con $s(n)$ intendiamo il successore di n.

La strategia che useremo per dimostrare sentenze della forma $s : A$ (il cui significato è "l'oggetto "s" soddisfa la proprietà A") è la seguente:

- troviamo una regola R la cui conclusione corrisponde alla sentenza $s : A$;
- dimostriamo tutte le precondizioni con la stessa strategia.

4.3 SEMANTICA STATICA

La semantica statica è il meccanismo che ci permette di descrivere proprietà di un programma che si manifestano senza doverlo eseguire. Queste proprietà sono spesso controllate dal compilatore o da strumenti esterni e ci permettono di avere un controllo statico sui programmi che vogliamo eseguire.

Per mostrare l'uso della semantica statica studiamo un semplice linguaggio di programmazione che può solo fare calcoli. La grammatica di questo linguaggio è la seguente:

```
E ::= Const | Ide | (Times E1 E2) | (Plus E1 E2) | (Let Ide E1 E2)
Const ::= 0 | 1 | 2 | ...
Ide ::= x | y | z | ...
```

Consideriamo il seguente judgment: E : ok se tutti gli identificatori contenuti in E sono definiti correttamente tramite un espressione di tipo Let.

Le regole per le costanti e per le operazioni di addizione e moltiplicazione sono molto semplici:

$$\frac{}{\texttt{Const : ok}} \qquad \frac{\texttt{E1 : ok} \quad \texttt{E2 : ok}}{\texttt{(Times E1 E2) : ok}} \qquad \frac{\texttt{E1 : ok} \quad \texttt{E2 : ok}}{\texttt{(Plus E1 E2) : ok}}$$

Tuttavia non possiamo al momento descrivere una regola per verificare la correttezza delle singole variabili e delle espressioni `Let`: data un'espressione del tipo `Ide` non abbiamo abbastanza informazione per decidere se essa è corretta oppure no, poiché potrebbe essere sia libera nell'espressione generale (e in tal caso sarebbe sbagliata), sia legata da qualche `Let` precedente.

Abbiamo quindi bisogno di introdurre una struttura di supporto per recuperare le informazioni relative agli identificatori. Chiameremo questa struttura **tabella dei simboli**, ed essa conterrà tutti i nomi delle variabili che abbiamo dichiarato nel programma. Inoltre se Γ è una tabella dei simboli, il judgment $\Gamma \vdash \texttt{e : A}$ signfica che considerando l'*ambiente* Γ l'espressione `e` ha la proprietà `A`.

Possiamo quindi esprimere completamente la condizione di correttezza di un programma del nostro linguaggio:

$$\frac{}{\Gamma \vdash \texttt{Const : ok}}$$

$$\frac{\Gamma \vdash \texttt{E1 : ok} \quad \Gamma \vdash \texttt{E2 : ok}}{\Gamma \vdash \texttt{(Times E1 E2) : ok}}$$

$$\frac{\Gamma \vdash \texttt{E1 : ok} \quad \Gamma \vdash \texttt{E2 : ok}}{\Gamma \vdash \texttt{(Plus E1 E2) : ok}}$$

$$\frac{x \in \Gamma}{\Gamma \vdash \texttt{x : ok}}$$

$$\frac{\Gamma \vdash \texttt{E1 : ok} \quad \Gamma \cup \{x\} \vdash \texttt{E2 : ok}}{\Gamma \vdash \texttt{(Let x E1 E2)}}$$

Simulazione in OCaml

Cerchiamo ora di simulare le regole dell'analisi statica in OCaml. Introduciamo innanzitutto le definizioni di tipo per le espressioni e gli identificatori:

```
type ide = string;;
type expr =
    | Int   of int
    | Den   of ide * int
    | Plus  of expr * expr
    | Times of expr * expr
    | Let   of ide * expr * expr;;
```

Dato un ambiente e un identificatore, controlliamo se l'identificatore esiste nell'ambiente:

```
let rec lookup st x =
    match st with
    | []    -> false
    | y::ys -> if x = y then true
                        else lookup ys x;;
```

La funzione che controlla se l'espressione è ben formata è quindi la seguente:

```
let rec check (e : expr) (st : string list) =
    match e with
    | Int n -> true
    | Den id -> lookup st x
    | Plus (e1, e2) -> (check e1 st) && (check e2 st)
    | Times (e1, e2) -> (check e1 st) && (check e2 st)
    | Let (id, e1, e2) -> (check e1 st) && (check e2 (x::st))
```

4.4 SEMANTICA DINAMICA

Vogliamo ora studiare la **semantica dinamica**, che ci dà le regole di esecuzione di un programma e ci permette di calcolarne il risultato. Vi sono diversi stili di semantica (in particolare *denotazionale*, *operazionale* e *assiomatica*), ma noi ci concentreremo sulla semantica operazionale ed in particolare su due tipi particolari:

- la **Structural Operational Semantics**, anche chiamata *semantica Small Steps*, che descrive ogni passo dell'esecuzione di un programma;
- la **Natural Semantics**, che descrive il risultato dell'esecuzione di un programma completo.

Semantica Small Steps

L'idea di base della semantica small steps è che ogni passo di valutazione ci porta da un'espressione ad un'espressione più semplice, fino ad arrivare ad un'espressione che non può più essere semplificata (un valore).

Si dice quindi **sistema di transizione** una tupla $(S, I, F, \rightarrow)$ dove:

- S è l'insieme dei possibili stati della macchina astratta del linguaggio;
- I è l'insieme degli stati iniziali;
- F è l'insieme degli stati finali;
- $\rightarrow \subseteq S \times S$ è la *relazione di transizione*, che descrive l'effetto di un singolo passo di valutazione.

Nel caso del nostro semplice linguaggio possiamo definire:

- S come l'insieme delle espressioni aritmetiche sintatticamente corrette (in qualche ambiente), ovvero
$$S \coloneqq \{\, \texttt{e} : \exists \Gamma \text{ tale che } \Gamma \vdash \texttt{e : ok} \,\};$$
- I come l'insieme delle espressioni aritmetiche "chiuse", ovvero che non contengono variabili libere:
$$I \coloneqq \{\, \texttt{e} : \varnothing \vdash \texttt{(e : ok)} \,\};$$
- F come l'insieme dei valori interi:
$$F \coloneqq \{\, \texttt{Int n} : n \in \mathbb{N} \,\}.$$

Iniziamo a studiare le regole di valutazione.

TIMES

Facciamo l'esempio della regola del costrutto `Times`:

$$\frac{}{\texttt{(Times (Int n) (Int m))} \rightarrow \texttt{Int (n * m)}}$$

$$\frac{e1 \rightarrow e1'}{\texttt{(Times e1 e2)} \rightarrow \texttt{(Times e1' e2)}}$$

$$\frac{e2 \rightarrow e2'}{\texttt{(Times (Int n) e2)} \rightarrow \texttt{(Times (Int n) e2')}}$$

Questa regola composta ci dice che

- se stiamo cercando di calcolare il risultato di `Times` applicato a due valori, il risultato è un valore (in particolare è `Int (n * m)`);
- se entrambi gli operandi non sono valori, eseguiamo uno step sul primo operando (che quindi può diventare un valore, ma può anche dover essere semplificato ancora) e rivalutiamo l'espressione;
- se il primo operando è un valore ma il secondo non lo è, eseguiamo uno step sul secondo operando e rivalutiamo l'espressione.

Facciamo alcune osservazioni:

- questa regola di moltiplicazione è **eager**: prima valuta completamente entrambi gli operandi, poi valuta l'espressione completa;
- inoltre anche l'ordine in cui vengono eseguite le valutazioni è evidente: prima viene eseguita la valutazione del primo operando, poi viene eseguita la valutazione del secondo.

LET

Studiamo ora la regola di valutazione del `Let`:

$$\frac{}{\texttt{(Let x (Int n) e2)} \to \texttt{e2[x := (Int n)]}}$$

$$\frac{e1 \to e1'}{\texttt{(Let x e1 e2)} \to \texttt{(Let x e1' e2)}}$$

Questa regola ci dice che

- se abbiamo valutato completamente `e1` al valore `(Int n)` allora lo step consiste nel restituire l'espressione `e2` dove tutte le occorrenze di `x` vengono sostituite da occorrenze di `(Int n)`;
- altrimenti eseguiamo uno step di valutazione sull'espressione `e1` e rivalutiamo l'espressione.

Semantica naturale

Nel caso della semantica naturale non siamo più interessati a semplificare un'espressione *un passo alla volta*, ma vogliamo valutare in un passo solo un'intera espressione. Per far ciò avremmo bisogno di

- un insieme E di espressioni valutabili,
- un insieme V di valori (che non è necessariamente un sottoinsieme di E),
- una relazione $\Rightarrow \subseteq \mathsf{E} \times V$ (anche indicata con $\Downarrow$), detta *relazione di valutazione*.

Ad esempio nel caso del nostro semplice linguaggio possiamo dare una semantica naturale nel seguente modo:

- E è l'insieme di tutte le espressioni ben formate, ovvero

$$\mathsf{E} = \{\, \texttt{e} : \Gamma \vdash \texttt{(e : ok)} \,\};$$

- V è l'insieme dei valori interi,
- alcune regole di valutazione sono le seguenti:

$$\frac{}{\texttt{Int n} \Rightarrow \texttt{Int n}}$$

$$\frac{\texttt{e1} \Rightarrow \texttt{Int n} \quad \texttt{e2} \Rightarrow \texttt{Int m}}{\texttt{Times e1 e2} \Rightarrow \texttt{Int (n*m)}}$$

$$\frac{\texttt{e1} \Rightarrow \texttt{(Int n)} \quad \texttt{e2[x := (Int n)]} \Rightarrow \texttt{(Int m)}}{\texttt{(Let x e1 e2)} \Rightarrow \texttt{Int m}}$$

Osserviamo che la regola data sopra per il `Let` è una regola di valutazione *eager*, nel senso che valuta tutti gli operandi prima di passare alla valutazione dell'operazione principale. Una regola di valutazione *lazy* per il `Let` è ad esempio la seguente:

$$\frac{\texttt{e2[x := e1]} \Rightarrow \texttt{(Int m)}}{\texttt{Let x e1 e2} \Rightarrow \texttt{(Int m)}.}$$

Tuttavia per definire correttamente un interprete abbiamo bisogno di un metodo per recuperare a tempo di esecuzione le informazioni relative alle variabili legate dai `Let` (e successivamente dalle funzioni): definiamo quindi il concetto di *ambiente* e di *binding*.

Si dice **binding** un'associazione tra un nome (di variabile) e un valore; si dice **ambiente** una funzione

$$\text{env} : \texttt{Ide} \longrightarrow \texttt{Values} \cup \{\, \texttt{Unbound} \,\}$$

dove `Ide` è l'insieme degli identificatori di variabili, `Values` è l'insieme dei valori e `Unbound` è il valore particolare assunto da una variabile che non è legata ad alcun valore. Per aggiungere un nuovo binding ad un ambiente useremo la notazione $\text{env}\,[x := v]$, che indica la funzione che si comporta esattamente come env per ogni $y \in \texttt{Ide} \setminus \{\, x \,\}$, mentre $\text{env}(x) = v$.

Una possibile prima implementazione di questo tipo di ambiente in OCaml è la seguente:

```
let empty_env = [];;

let rec lookup env x =
    match env with
    | []         -> failwith ("not found")
    | (a, v)::as -> if x = a then v
                             else lookup as x;;

let bind x v env = (x, v)::env;;
```

Possiamo quindi specificare meglio la nostra costruzione della semantica naturale: abbiamo bisogno di

- un insieme di stati

  $$S = \{\, \rho \triangleright e \;:\; \rho \in \texttt{Env},\ e \in \texttt{Exp},\ \varnothing \vdash (e \;:\; \texttt{ok}) \,\}.$$

  Ogni stato rappresenta una coppia formata da un ambiente e da un'espressione da valutare in quell'ambiente.

- un insieme di valori, che nel nostro caso sono i numeri naturali,
- una relazione di valutazione $\Rightarrow \subseteq S \times V$.

Le regole di valutazione dell'interprete diventano quindi ad esempio:

$$\frac{\text{env}(\texttt{id}) = v \in V}{\text{env} \triangleright \texttt{(Den id)} \Rightarrow v}$$

$$\frac{\text{env} \triangleright e_1 \Rightarrow v_1 \quad \text{env} \triangleright e_2 \Rightarrow v_2 \quad v_1 \cdot v_2 = v}{\text{env} \triangleright \texttt{(Times } e_1\ e_2\texttt{)} \Rightarrow v}$$

$$\frac{\text{env} \triangleright e \Rightarrow v' \quad \text{env}\,[x := v'] \triangleright \texttt{body} \Rightarrow v}{\text{env} \triangleright \texttt{(let } x = e \texttt{ in body)} \Rightarrow v}$$

Una possibile ottimizzazione di questo processo viene data dall'uso degli **indici di De Bruijn**: a tempo di compilazione sostituiamo il nome di ogni variabile con il numero di `Let` che bisogna attraversare per raggiungere il `Let` in cui è stata definita tale variabile. Quindi ad esempio l'espressione

```
Let ("x", (Int 5), (Let ("z", (Int 17), (Times (Den "z", Den "x"))))
```

può essere trasformata nell'espressione

```
Let ((Int 5), (Let ((Int 17), (Times (Den 0, Den 1)))))
```

4.5 TIPI DI DATO

Finora il nostro semplice linguaggio può produrre soltanto valori di tipo intero, quindi non vi è la necessità di un sistema di tipi. Tuttavia quando i valori cominciano ad essere più complicati, un **type system** può essere utile per diversi motivi:

- i tipi sono utili a livello di *progetto*: organizzano l'informazione e ci consentono di astrarre esplicitamente sui dati;
- sono utili a livello di *programma*: identificano e prevengono alcuni errori automaticamente;
- sono anche utili a livello di *implementazione*: tipi diversi richiedono risorse diverse (ad esempio un booleano richiede solo un bit, mentre un valore floating-point ad alta precisione ne richiede 64) e quindi forniscono alcune informazioni necessarie alla macchina astratta per allocare lo spazio di memoria.

Distinguiamo innanzitutto tra diversi tipi di dati:

- un dato si dice *denotabile* se può essere associato ad un nome;
- un dato si dice *esprimibile* se può essere il risultato della valutazione di un'espressione;
- un dato si dice *memorizzabile* se può essere memorizzato in una variabile.

Ad esempio le funzioni in OCaml sono denotabili ed esprimibili ma non sono memorizzabili, mentre in C sono solamente denotabili.

Descrittori di dato

Quando studiamo i tipi di dato vogliamo studiare sia la loro semantica, sia la loro implementazione. Partendo da quest'ultima, sembra necessario che un tipo di dato nella sua rappresentazione concreta contenga una "descrizione del tipo": a run-time vogliamo infatti (ad esempio) essere certi che il tipo del dato che abbiamo sia quello previsto dall'operazione che stiamo effettuando.

In OCaml potremmo rappresentare questo fatto nel seguente modo:

```
(* Espressione sintattica *)
type exp =
    | EInt of int
    | EBool of bool

(* Tipo a run-time *)
type evT =
    | Int of int
    | Bool of bool

(* Funzione per il typechecking *)
let typecheck descr x =
    match descr with
    | "int" -> (match x with
                | Int n -> true
                | _     -> false)
    | "bool" -> (match x with
                 | Bool b -> true
                 | _      -> false)
```

Tuttavia l'uso dei descrittori di dato può essere superfluo a seconda del tipo di linguaggio considerato.

- Se l'informazione sui tipi è conosciuta completamente a tempo di compilazione (come nel caso di OCaml) si possono eliminare i descrittori di dato, in quanto il typecheck è effettuato dal compilatore (*typecheck statico*).
- Se l'informazione sui tipi è conosciuta solamente a tempo di esecuzione (come ad esempio in Javascript) i descrittori sono necessari per tutti i tipi e il typechecking è completamente *dinamico*.
- Se l'informazione sui tipi è conosciuta parzialmente a tempo di compilazione (come nel caso di Java) i descrittori di dato devono contenere solo l'informazione "dinamica" e il typecheck è effettuato parzialmente a tempo di compilazione e parzialmente a tempo di esecuzione.

I tipi predefiniti vengono anche chiamati *tipi scalari*. Tra essi vi sono:

- gli interi,
- i booleani,
- i caratteri,
- i numeri reali (floating point),
- il tipo `void`, rappresentato in OCaml da `()`, con le seguenti caratteristiche:
  - ha un solo valore,
  - non ha operazioni,
  - serve per implementare operazioni che modificano lo stato senza restituire un vero valore.

Tuttavia i linguaggi definiscono anche dei **tipi composti**. Ve ne sono diversi:

- i record (chiamati `struct` in C),
- i record varianti, oppure *sum types* (in cui solo un tipo è attivo in un dato istante di tempo, chiamati `union` in C),
- gli array,
- gli insiemi,
- i puntatori.

Record

I record sono stati definiti per manipolare in modo unitario dati di tipo eterogeneo. Ad esempio il tipo

```
struct studente {
    int matricola;
    char nome[20];
}
```

definisce un tipo di dato che rappresenta uno studente, con il suo nome e numero di matricola.

L'implementazione di un record viene fatta *sequenzialmente*: i vari campi occupano posti consecutivi in memoria. In particolare abbiamo due possibili strategie:

- allineamento alla parola;
- packed record.

Nel caso di *allineamento alla parola* ogni campo del record deve occupare una o più parole intere, dunque se occupa meno spazio di una parola (tipicamente 4 byte) allora viene lasciato dello spazio libero (chiamato *padding*). Questo porta ad uno spreco di memoria, ma contemporaneamente permette di avere degli accessi molto semplici ai campi (si trovano tutti all'inizio di una parola di memoria).

Nel caso di *packed record* i campi vengono scritti consecutivamente, senza lasciare spazio tra un campo e l'altro e quindi senza rispettare l'allineamento alla parola. L'accesso ai campi si fa più complicato, tuttavia non vi è spreco di memoria.

Simulazione dei record in OCaml

Per simulare il comportamento dei record, estendiamo la sintassi astratta del nostro linguaggio con i seguenti costruttori:

```
type label = Lab of string
type expr = ...
    | Record of (label * expr) list
    | Select of label * expr
```

Possiamo quindi dichiarare record della seguente forma:

```
Record [(Lab "size", Int 5), (Lab "weight", Int 255)]
```

Per interpretare la creazione di un record e l'operazione di selezione dobbiamo innanzitutto estendere i tipi esprimibili a run-time dal linguaggio:

```
type evT = ...
    | RecordEv of (label * evT) list
```

A questo punto introduciamo una funzione trovare un valore in un `RecordEv` data una `label`:

```
let rec lookupRecord (body : (label * evT) list) (lab : label) : evT =
    match body with
    | [] -> raise FieldNotFound
    | (lab', value)::rs -> if lab' = lab then value
                           else lookupRecord rs lab
```

Possiamo quindi estendere la funzione di valutazione con la valutazione dei due nuovi campi:

```
let rec eval (exp : expr) : evT =
    match exp with
    | ...
    | Record recordBody -> RecordEv (evalRecord recordBody)
    | Select(e, lab) -> (match eval e with
                         | RecordEv body -> lookupRecord body lab
                         | _ -> raise TypeMismatch)
and evalRecord (body : (label * expr) list) : (label * evT) list =
    match body with
    | [] -> []
    | (lab, e)::rs -> (lab, eval e)::evalRecord rs
```

Array

Un **array** è una collezione di dati omogenei. In particolare, astrattamente un array è una funzione da un insieme di indici (solitamente di tipo intero) ad un insieme di elementi (di tipo qualunque, ma solitamente non funzionale).

La principale operazione ammessa sugli array è la selezione di un elemento: la modifica infatti non è propriamente un'operazione sull'array ma piuttosto un'operazione sulla locazione di memoria che memorizza quel dato.

Gli array vengono memorizzati in locazioni continue di memoria: nel caso degli array multidimensionali (le matrici) si può sfruttare sia un *ordine di riga* in cui vengono memorizzate le righe in ordine, oppure un *ordine di colonna*; solitamente il primo è più usato.

Per accedere agli elementi di un array si può usare una semplice formula: dato l'indirizzo della base b dell'array e un offset i, per selezionare l'elemento i-esimo basta calcolare

$$b + c \cdot i,$$

dove c è la dimensione in byte degli elementi contenuti nell'array. Esiste una formula analoga (ma più complicata) per gli array multidimensionali.

Puntatori

I **puntatori** sono dei riferimenti a delle locazioni di memoria, oppure la costante `NULL`.

4.6 TYPE-SYSTEM

Cerchiamo ora di comprendere come deve essere implementato un *type-system* per un linguaggio di programmazione.

Un **type system** è una funzione che associa ad ogni oggetto un *tipo*. Esaminando il flusso dei valori calcolati, il sistema ci assicura che il programma non contenga errori di tipo.

Per tenere traccia del tipo di ogni identificatore nel nostro programma dobbiamo usare un **ambiente dei tipi**. Indicheremo con

$$\Gamma = x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n$$

l'ambiente dei tipi che associa all'espressione x_i il valore τ_i. Inoltre useremo il simbolo

$$(\Gamma,\ x : \tau)$$

per indicare l'ambiente dei tipi che associa x al tipo τ e associa ogni espressione y diversa da x il tipo ad essa associato precedentemente da Γ. Infine, il judgment

$$\Gamma \vdash x : \tau$$

indica che nell'ambiente dei tipi Γ l'espressione x ha tipo τ.

Definiamo ora un semplice sistema dei tipi per le nostre espressioni.

$$\Gamma \vdash n : \texttt{int}$$

$$\Gamma \vdash b : \texttt{bool}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash \texttt{e1} : \tau' \quad (\Gamma, x : \tau') \vdash \texttt{e2} : \tau}{\Gamma \vdash \texttt{let x = e1 in e2} : \tau}$$

$$\frac{\Gamma \vdash \texttt{e1} : \tau_1 \quad \Gamma \vdash \texttt{e2} : \tau_2 \quad \Gamma \vdash \otimes : \tau_1 \times \tau_2 \to \tau}{\Gamma \vdash \texttt{e1} \otimes \texttt{e2} : \tau}$$

$$\frac{\Gamma \vdash \texttt{cond} : \texttt{bool} \quad \Gamma \vdash \texttt{e1} : \tau \quad \Gamma \vdash \texttt{e2} : \tau}{\Gamma \vdash \texttt{(if cond then e1 else e2)} : \tau}$$

Nel caso delle funzioni il type system deve diventare necessariamente più complicato. Definiamo il tipo di una funzione come il *tipo freccia* $\tau_1 \rightarrow \tau_2$, dove τ_1, τ_2 sono due tipi del nostro type system.

Abbiamo quindi le seguenti due regole di inferenza, una per le definizioni di funzione e una per le applicazioni:

$$\frac{(\Gamma, \texttt{x} : \tau_1) \vdash \texttt{e} : \tau_2}{\Gamma \vdash \texttt{fun (x : } \tau_1\texttt{) = e} : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash \texttt{e1} : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash \texttt{e2} : \tau_1}{\Gamma \vdash \texttt{App(e1, e2)} : \tau_2}$$

Queste due regole possono tuttavia risultare insufficienti nel caso di *scoping dinamico*. Parleremo in seguito del concetto di scoping statico e dinamico, ma per il momento accontentiamoci di questo esempio.

```
let x = 10 in
let f = fun k -> k + x in
let x = 5 in
    f 10;;
```

Se analizziamo questo codice abbiamo due possibili interpretazioni:

- la x contenuta nella chiamata a funzione `f x` si riferisce alla variabile x legata al valore 10 dal primo operatore **`let`**: infatti essa è l'unica variabile chiamata x al momento della definizione della funzione;
- la x contenuta nella chiamata a funzione `f x` si riferisce alla variabile x legata al valore 5 dal terzo operatore **`let`**.

Nel primo caso (chiamato *scoping statico*) il valore dell'espressione complessiva è 20; nel secondo (chiamato *scoping dinamico*) il valore è invece 15. Tuttavia nel secondo caso le regole del nostro type-system non possono essere controllate staticamente: l'espressione

```
let x = 10 in
let f = fun k -> k + x in
let x = false in
    f 10;;
```

conterrebbe un ovvio errore di tipo che non si manifesterebbe fino al momento dell'esecuzione.

4.6.1 Implementazione in OCaml di un type system

Innanzitutto dobbiamo decidere quali sono i tipi del nostro linguaggio:

```
type tval =
    | TInt
    | TBool
    | FunT of tval * tval
```

A questo punto dobbiamo creare l'ambiente dei tipi:

```
type tenv = ide -> tval
```

Le funzioni `bind_t` e `empty_tenv` ci consentono rispettivamente di aggiungere un binding ad un ambiente dei tipi dato e di ottenere l'ambiente vuoto.

```
let bind_t (env : tenv) (id : ide) (v : tval) : tenv =
    fun x -> if x = id then v else env x
let empty_tenv : tenv = fun (x : ide) -> raise EmptyEnv
```

Per poter sfruttare il typechecker dobbiamo modificare il tipo delle espressioni che usiamo, aggiungendo dove necessario i tipi dei parametri.

```
type texp =
    | EInt of int
    | EBool of bool
    | Den of ide
    | Add of texp * texp
    | ...
    | Let of ide * texp * texp
    | Fun of ide * tval * texp
    | Apply of texp * texp
```

Osserviamo che l'unica modifica fatta è al costruttore delle funzioni anonime: in questo caso dobbiamo sapere di che tipo è il parametro per poter dedurre il tipo della funzione.

Il typechecker rispetta precisamente le regole semantiche date in precedenza:

```
let rec teval (e : texp) (env : tenv) : tval =
    match e with
    | EInt a -> TInt
    | EBool b -> TBool
    | Den s -> env s
    | Let (id, e1, e2) ->
      let t = teval e1 tenv in
          teval e2 (bind tenv id t)
    | Add (e1, e2) ->
      let t1 = teval e1 tenv in
      let t2 = teval e2 tenv in
          ( match t1, t2 with
              | (TInt, TInt) -> TInt
              | _ -> raise WrongType)
    | ...
    | If (cond, e1, e2) ->
      let tcond = teval cond tenv in
          ( match tcond with
              | TBool ->
                  let t1 = teval e1 tenv in
                  let t2 = teval e2 tenv in
                      ( match t1, t2 with
                          | TInt, TInt -> TInt
                          | TBool, TBool -> TBool
                          | _ -> raise WrongType ))
    | Fun (id, tpar, body) ->
      let tenv1 = bind tenv id tpar in
      let t2 = teval body tenv1 in
          FunT (t1, t2)
    | Apply (e1, e2) ->
      let f = teval e1 tenv
          ( match f with
              | Fun(t1, t2) ->
                  if (teval e2 tenv = t1) then t2
                  else raise WrongType
              | _ -> raise WrongType )
```

Possiamo anche estendere il type system definito in precedenza con il supporto ad altri tipi, ad esempio con il supporto alle funzioni ricorsive, oppure alle coppie di valori, eccetera.

4.7 CHIAMATE A FUNZIONE

Nel nostro interprete non abbiamo ancora definito una sintassi e una semantica per la dichiarazione e l'applicazione di funzioni. La dichiarazione di funzione ci pone già diversi problemi:

- come descrivo il passaggio dei parametri?
- se nella funzione ci sono variabili dichiarate esternamente alla funzione (variabili *non-locali*), dove trovo il loro valore?

Per questo conviene analizzare prima un costrutto più semplice, ovvero i **blocchi**.

4.7.1 Blocchi

Un blocco è una procedura senza nome e senza parametri. Ad esempio nel C i blocchi vengono realizzati attraverso l'uso di parentesi graffe:

```
{ // starting point of the first block
    int x = 5;
    { // starting point of the second block
        int y = 2;
        int z = x + y;
    } // ending point of the second block
    x = x + 2;
} // ending point of the second block
```

Il sistema per gestire i blocchi (e, come vedremo in seguito, le funzioni) viene chiamato **stack dei record di attivazione**. Un record di attivazione è semplicemente una porzione riservata di memoria che contiene spazio sufficiente per memorizzare delle variabili, più dello spazio aggiuntivo per ulteriori dati che esamineremo più avanti.

I record di attivazione vengono gestiti tramite uno stack in quanto è la struttura dati più adatta allo scopo. Nel caso precedente, ad esempio, lo stack dei record di attivazione subisce le seguenti modifiche:

1. All'inizio lo stack S è vuoto.
2. Quando entriamo nel primo blocco viene creato un nuovo record di attivazione R_1, contenente lo spazio per la variabile intera x; questo record viene immediatamente inserito nello stack con un'operazione di *push*.
3. All'entrata nel secondo blocco verrà creato un nuovo record di attivazione R_2 contenente lo spazio per y e z; anch'esso viene immediatamente inserito nello stack con un'operazione di push.
4. Nell'inizializzare z si usa il valore di x contenuto nel record R_1: x è un riferimento *non-locale* nel record attuale.
5. Alla fine del blocco interno si rimuove il record di attivazione dallo stack con una *pop*. Da questo punto in poi le variabili y e z non sono più allocate.
6. Alla fine del blocco esterno si rimuove il record R_1 tramite una *pop*.

Osserviamo che dobbiamo prevedere dello spazio in ogni record per memorizzare i risultati intermedi, altrimenti non potremmo svolgere calcoli complessi. Inoltre abbiamo bisogno di un meccanismo per risalire da un record ad un altro record posizionato *più in alto* nella catena di record contenuti nello stack: questo meccanismo ci è dato dal **control link**.

Un record di attivazione per un blocco è quindi formato da 3 parti:

- un **control link** (o *puntatore di catena dinamica*) che rappresenta il puntatore dal blocco corrente al record appena precedente nello stack;
- dello spazio per le variabili di istanza;
- dello spazio per i risultati temporanei.

Ogni volta che facciamo una push di un record di attivazione sullo stack dobbiamo aggiornare il control link del blocco corrente per farlo puntare al blocco precedente; ogni volta che facciamo una push ci basterà distruggere questo puntatore. Inoltre il puntatore al record in cima allo stack viene chiamato **environment pointer** o **stack pointer** e viene aggiornato ogni volta che aggiungiamo o rimuoviamo un record dallo stack.

Regole di scope nel caso dei blocchi

Nell'esempio precedente scritto in C abbiamo notato che abbiamo bisogno di un meccanismo per calcolare il valore di riferimenti non locali. I due meccanismi vengono chiamati **scoping statico** o **scoping dinamico**:

- nel caso di scoping statico i riferimenti non locali vengono risolti nel blocco più vicino *sintatticamente* nella struttura del programma;
- nel caso di scoping dinamico i riferimenti non locali vengono risolti nel record di attivazione precedente nell'*esecuzione* del programma.

Fortunatamente nel caso dei blocchi i due metodi sono equivalenti: possiamo considerare l'uno o l'altro senza distinzioni.

4.7.2 Funzioni e procedure

Anche nel caso di funzioni e procedure si usano i record di attivazione: in questo caso il record deve contenere spazio almeno per:

- il control link;
- i parametri della funzione;
- l'indirizzo di ritorno;
- le variabili locali;
- i risultati intermedi;
- il valore restituito (anche se possiamo considerarlo come un risultato intermedio);
- indirizzo in cui inserire il valore restituito.

Passagio dei parametri

Sappiamo che esistono diverse modalità per il passaggio dei parametri:

- nel caso del *passaggio per valore*, implementato ad esempio dal C e dal Java, nel record di attivazione viene copiato il valore del parametro: la funzione non può modificare il contenuto del parametro e non vi è *aliasing* (più variabili che si riferiscono alla stessa locazione di memoria).
- nel caso del *passaggio per riferimento* viene copiato nel record di attivazione l'*indirizzo* del parametro attuale: si crea quindi una situazione di aliasing, da cui segue che la funzione può modificare il contenuto della variabile.

Bisogna prestare attenzione al fatto che nel caso del C e del Java l'unica modalità di passaggio dei parametri consentita è il passaggio per valore: infatti quando si passa un puntatore/riferimento ad un oggetto in realtà si sta copiando il valore della variabile, soltanto che questo valore è una locazione di memoria.

Regole di scope nel caso di funzioni

Consideriamo ora il seguente esempio:

```
var x = 1;
function g(z) {
    return x + z;
}
function f(y) {
    x = y+1;
    return g(y*x);
}
f(3);
```

Qual è il valore di `f(3)` nel caso di scoping statico o dinamico?

- Nel caso di scoping statico la variabile `x` contenuta nella funzione `g` si riferisce alla variabile dichiarata nella prima linea di codice. Dunque nell'esecuzione della funzione i passi di calcolo saranno:
  - nel calcolo di `f(4)` si pone `x` uguale a `4` e si calcola `g(3*4)`;
  - nel calcolo di `g(12)` si usa il valore di `x` definito staticamente, ovvero `x = 1`, e si restituisce quindi `13`, che è quindi il valore complessivo della chiamata a funzione `f(3)`.
- Nel caso di scoping dinamico ogni riferimento non locale viene risolto *dinamicamente*, durante l'esecuzione del programma. Dunque:
  - nel calcolo di `f(4)` si pone `x` uguale a `4` e si calcola `g(3*4)`;
  - nel calcolo di `g(12)` si usa il valore di `x` contenuto nel record di attivazione precedente, che è il record che contiene la chiamata a `f(3)`: segue quindi che `x` vale 4 e il risultato della funzione è `16`.

Nel caso delle chiamate a funzione quindi i due meccanismi sono diversi e portano a comportamenti diversi: i linguaggi di programmazione devono quindi scegliere se adottare lo scoping statico (scelta fatta dalla maggior parte dei linguaggi) o lo scoping dinamico (scelta fatta principalmente dal LISP e da alcuni linguaggi di scripting).

Scoping statico

Analizziamo ora più nel dettaglio il funzionamento dello scoping statico: il record di attivazione di una funzione nel caso di scoping statico deve contenere, in aggiunta a quanto detto precedentemente, uno **static link** che collega il blocco corrente al blocco padre nella gerarchia statica.

I due *link* (quello statico e quello dinamico) hanno quindi ruoli diversi: il primo viene usato per risolvere i riferimenti non locali ed è determinato staticamente, senza dover eseguire il codice, mentre il secondo è determinato solamente dall'ordine delle chiamate di funzione e quindi ha senso soltanto durante l'esecuzione del programma.

Esempio 4.7.1. Supponiamo di aver a che fare con il seguente codice simil-C:

```
{   // MAIN
    int x;
    void A() {
        x = x+1;
    }
    void B () {
        int x;
        void C (int y) {
            int x;
            x = y+2;
            A ();
        }
        x = 0;
        A();
        C(3);
    }
    x = 10;
    B();
}
```

Siccome le funzioni `A` e `B` sono dichiarate nel blocco "main", ogni volta che verranno chiamate avremo che il loro *static link* punterà al blocco main; invece la funzione `C` avrà uno static link che punterà al blocco della funzione `B` dove è stata dichiarata.

Il risultato dell'esecuzione del codice è dato dalla Figura 4.1 (le frecce tratteggiate sono i puntatori di catena statica):

- nel main x viene inizializzato a 12 e viene chiamata la funzione `B();`
- la funzione dichiara una nuova variabile x, che viene inizializzata a 0, dopo chiama la funzione `A();`
- `A` deve calcolare `x = x+1`, ma siccome x è un riferimento non locale deve cercarlo seguendo il puntatore di catena statica: la x trovata è quella dichiarata nel main, che vale 12. Il programma pone quindi la x del main uguale a 13 e ritorna;
- continua l'esecuzione della funzione `B`, che chiama a questo punto `C(3);`
- la funzione `C` dichiara una nuova variabile x e la pone uguale a $y + 2$; siccome y è un riferimento locale il valore della x locale diventa quindi 5. Subito dopo `C` chiama la funzione `A`;
- ancora una volta `A` deve trovare il riferimento non locale x seguendo il puntatore di catena statica e, come nel caso precedente, trova il valore dichiarato nel main e lo aggiorna a 14;
- si conclude l'esecuzione del programma.

Funzioni come valori e chiusure

Nel caso dei linguaggi funzionali le funzioni tuttavia sono dei valori: possono essere passate ad altre funzioni e possono essere restituite come risultato di funzioni. Consideriamo ad esempio il seguente codice:

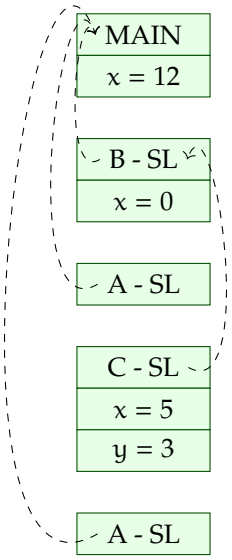

FIGURA 4.1: Struttura dei record di attivazione

```
let x = 4;
let f(y) = x*y;
let g(h) = let x = 7 in
           h(3) + x;
g(f);
```

La funzione `f` prende un parametro e lo moltiplica per x; la funzione `g` prende una funzione `h` come parametro, dichiara x = 7 e restituisce il valore della funzione `h` valutata nel punto 3 più il valore di x.

La domanda che possiamo porci è: quale definizione di x deve essere usata nella chiamata `g(f)`?

Assumendo regole di scoping statico la chiamata `h(3)` deve usare la definizione di `f` data poco sopra, quindi il valore della x deve essere 4, mentre quando sommiamo x al risultato di `h(3)` la x in questione è quella legata dal **`let`**, quindi deve essere uguale a 7.

Quindi nel chiamare la funzione `f` dobbiamo tener traccia sia del suo codice (cioè ciò che la funzione fa) sia dell'ambiente in cui è stata dichiarata (memorizzato dal puntatore di catena statica), in modo da poter risolvere eventuali riferimenti non locali.

Il valore di una funzione è quindi una coppia

$$\langle \text{codice funzione}, \text{ambiente di dichiarazione} \rangle.$$

Una tale coppia si chiama **chiusura** (o in inglese **closure**).

Quando una funzione viene invocata si allocal sullo stack il suo record di attivazione, si va a considerare la chiusura relativa alla funzione e si usa come *puntatore di catena statica* il puntatore all'ambiente di dichiarazione, come possiamo vedere in Figura 4.2 per quanto riguarda il programma precedente.

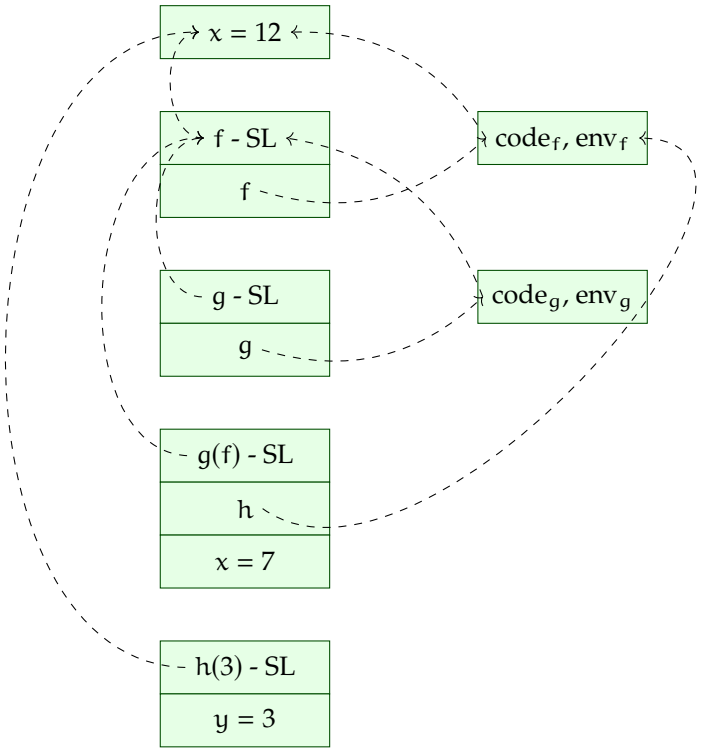

FIGURA 4.2: Struttura dei record di attivazione con le chiusure

CHIUSURE RICORSIVE

Nel caso di funzioni ricorsive l'ambiente di dichiarazione non è l'ambiente puntato dallo static link, ma è la funzione stessa: infatti se la funzione è ricorsiva il nome della funzione stessa deve essere nell'ambiente di dichiarazione.

CHIUSURE COME VALORI DI RITORNO

L'ultimo caso è quello in cui vogliamo restituire una funzione: in questo caso il valore restituito è una chiusura $\langle$`env`, `code`$\rangle$, ma dobbiamo segnalare che l'ambiente di dichiarazione `env` non può essere distrutto fino a quando la funzione può essere usata. Questo meccanismo si chiama **retention**.

FUNZIONI IN AMBIENTE DINAMICO

Se il linguaggio sfrutta lo scoping dinamico non è necessario usare i puntatori di catena statici (che non esistono), poiché le associazioni non locali vengono sempre risolte nel record di attivazione precedente in senso dinamico. In particolare non è neanche necessario usare le chiusure, poiché ci basta risalire la pila dei record di attivazione.

Implementazione dei record di attivazione

L'implementazione effettiva dei record di attivazione varia da linguaggio a linguaggio, ma ci sono delle linee guida generali.

Innanzitutto tutte le dichiarazioni vengono solitamente trasformate dal compilatore in una coppia $\langle$`livello`, `offset`$\rangle$:

- il `livello` indica il livello lessicale in cui è dichiarata la variabile, ovvero in quale blocco si trova;

- l'`offset` ci permette di distinguere la variabile dalle altre variabili dichiarate nello stesso blocco.

Le variabili nel codice vengono poi trasformate in un'altra coppia di valori: il secondo è l'`offset` dichiarato prima, mentre il primo rappresenta la differenza tra il livello in cui la variabile viene usata e il livello in cui la variabile viene dichiarata; ci dice quindi di quanto dobbiamo risalire la catena statica per trovare la variabile.

4.8 PASSAGGIO DEI PARAMETRI

Vogliamo ora studiare più in dettaglio le modalità di passaggio dei parametri nei vari linguaggi di programmazione.

Innanzitutto lo scopo dell'*astrazione parametrica* è quella di poter usare lo stesso frammento di codice con dati diversi a seconda della necessità: in particolare la *funzione* viene dichiarata con una lista di **parametri formali**, che rappresentano dei placeholder per i valori che verranno sostituiti in seguito. I valori passati alla funzione (chiamati **parametri attuali**) sono nella forma di una lista di espressioni: ognuno di essi verrà sostituito al posto del corrispondente parametro formale.

Il passaggio dei parametri è quindi una specie di *dichiarazione dinamica*: al momento della chiamata di funzione viene fatto un binding tra il parametro formale e il valore ottenuto dalla valutazione del parametro attuale. Esistono quindi diverse modalitàdi passaggio di parametri:

- passaggio per costante;
- passaggio per riferimento;
- passaggio di oggetti;
- passaggio di funzioni o procedura.

Passaggio per costante

Nel caso di passaggio per costante il valore passato deve essere un valore *immutable*: il sottoprogramma non può modificarlo. Questo tipo di passaggio è il tipo di tutti i linguaggi funzionali, ma è presente anche in alcuni linguaggi imperativi.

Passaggio per riferimento

In questo caso il parametro formale ha come tipo un valore modificabile: viene passata una locazione di memoria, quindi possiamo direttamente modificare il suo contenuto e gli effetti si ripercuotono all'esterno della procedura (fenomeno dell'**aliasing**). Spesso il passaggio per riferimento non è realizzato concretamente, ma simulato attraverso passaggio del *valore* di puntatori (come nel caso del C).

Passaggio di oggetti

In questo caso il parametro formale ha come tipo un *puntatore ad un oggetto*: non possiamo modificare il puntatore, ma attraverso esso possiamo modificare l'oggetto puntato.

Passaggio di funzioni

Il parametro formale ha come tipo una funzione, procedura o classe: l'espressione passata come parametro attuale deve essere il nome di una classe o anche un'espressione che ritorna una funzione, dunque la sua valutazione è una chiusura. Questo tipo particolare di parametro può essere solamente passato ulteriormente ad altre funzioni, oppure può essere attivato (la funzione può essere chiamata o la classe può essere istanziata tramite la `new`). Siccome nei linguaggi imperativi e object-oriented solitamente le funzioni non sono esprimibili, questo tipo di passaggio di parametri è presente principalmente nei linguaggi funzionali.

Esistono anche altre tecniche di passaggio di parametri, come

- **passaggio per valore** o **risultato**: in questi caso oltre all'ambiente si valuta anche la memoria;
- **passaggio per nome**: il parametro attuale non viene valutato.

Passaggio per valore

Questo meccanismo coinvolge valori modificabili e pertanto non esiste nei linguaggi funzionali puri: consiste nel creare una variabile con il nome del parametro formale ed assegnare ad essa il valore del parametro attuale. In questo modo dobbiamo quindi coinvolgere la memoria, in quanto scriviamo nella locazione occupata dal parametro formale il valore del parametro attuale. In particolare, se l'assegnamento è implementato correttamente l'ambiente non viene coinvolto, non viene creato aliasing e non si possono avere effetti laterali (cioè eventuali modifiche al parametro formale non si ripercuotono all'esterno della funzione).

Passaggio per valore-risultato

Nel caso si voglia modificare anche il parametro attuale modificando il parametro formalem, ma senza ricorrere agli effetti laterali diretti del passaggio per riferimento, si può implementare un meccanismo di passaggio per valore-risultato: come nel caso precedente quando la funzione viene chiamata si assegna il valore del parametro attuale alla variabile che rappresenta il parametro formale, ma alla fine della funzione viene anche fatto l'assegnamento inverso, per cui la variabile del parametro attuale viene modificata.

Esiste anche un passaggio per risultato, dove viene fatto solo il secondo assegnamento.

Osserviamo che il passaggio per valore-risultato è diverso dal passaggio per riferimento: infatti nel primo caso si crea una copia del valore del parametro attuale, per cui durante l'esecuzione della funzione i due parametri si riferiscono a due variabili (e quindi a due locazioni di memoria) diverse che possono evolvere indipendentemente; invece nel secondo caso le due variabili si riferiscono alla stessa locazione di memoria (*aliasing*) e quindi ogni modifica fatta al parametro formale si ripercuote immediatamente sul parametro attuale.

Passaggio per nome

Nel caso del passaggio per nome l'espressione passata in corrispondenza del parametro attuale non viene valutata immediatamente: verrà valutata ogni volta che si incontra (eventualmente) un'occorrenza del parametro formale durante l'esecuzione della funzione. L'espressione non valutata diventa quindi una *chiusura* contenente l'espressione e l'ambiente al momento della definizione: ogni volta che sarà necessario valutarla, dovremo farlo nell'ambiente di definizione.

Un'espressione passata per nome è quindi equivalente ad una funzione senza parametri: ogni volta che vogliamo valutarla dobbiamo fare lo stesso procedimento che seguiamo nel caso di un'applicazione di funzione. Una peculiarità delle espressioni passate per nome è che vanno sempre valutate nell'ambiente di definizione, anche nel caso di linguaggi con scoping dinamico.

Questo meccanismo di passaggio è presente in linguaggi come l'ALGOL e il LISP, è alla base del meccanismo di valutazione lazy di Haskell ed è simulabile in altri linguaggi funzionali (come OCaml) tramite funzioni senza parametri.

4.9 LINGUAGGIO DIDATTICO

In questa sezione ci occuperemo di implementare in OCaml un piccolo nucleo di un linguaggio funzionale con scoping statico, contenente

- interi e booleani,
- poche operazioni di base,

- il costrutto `if-then-else`,
- il costrutto `let-in`,
- funzioni e funzioni ricorsive.

Definiamo innanzitutto la sintassi astratta e i tipi esprimibili a runtime dal linguaggio.

```
(* ide represents a variable identifier *)
type ide = string

(* expr is the Abstract Syntax Tree of the language*)
type expr =
  (* base values *)
  | LitTrue
  | LitFalse
  | LitInt of int
  (* variables *)
  | Den of ide
  (* basic operation *)
  | Add     of expr * expr
  | Sub     of expr * expr
  | Times   of expr * expr
  | IsZero  of expr
  | Equals  of expr * expr
  (* if-then-else construct *)
  | IfThenElse of expr * expr * expr
  (* let-in construct *)
  | Let of ide * expr * expr
  (* anonymous function *)
  | Fun of ide * expr
  (* function application *)
  | Apply of expr * expr

(* Runtime values *)
type evT =
  | Int of int
  | Bool of bool
  | Unbound
```

Ricordiamo inoltre che è necessario definire un ambiente di valutazione, altrimenti non possiamo memorizzare i valori delle variabili usate.

```
(* The polymorphic environment *)
type 'v env = (string * 'v) list

(* empty_env returns an empty environment *)
let empty_env : evT env = [("", Unbound)]
(* bind adds a new binding to a pre-existing environment *)
let bind (env : evT env) (x : ide) (v : evT) = (x, v)::env
(* lookup looks for the value of the variable identified by x in the environment *)
let rec lookup (env : evT env) (x : ide) =
  match env with
  | []                    -> Unbound
  | (a, v)::_ when a = x  -> v
  | _::e                  -> lookup e x
```

Implementiamo anche una semplicissima forma di typechecking dinamico.

```
let typecheck (t : string) (v : evT) =
  match t with
  | "int" ->
    ( match v with
    | Int(_) -> true
    | _      -> false )
  | "bool" ->
    ( match v with
    | Bool(_) -> true
    | _       -> false )
  | _ -> failwith "not a valid type"
```

Possiamo dunque iniziare a definire la funzione di valutazione: innanzitutto sappiamo che dovrà prendere in input un'espressione, un ambiente di valori e dovrà restituire un valore. Avrà quindi la seguente forma:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
```

Iniziamo ad aggiungere mano mano le varie espressioni possibili all'interprete.

Valori letterali

Valutare letterali è ovvio.

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | LitInt(n) -> Int n
  | LitTrue   -> Bool true
  | LitFalse  -> Bool false
```

Variabili

Per valutare una variabile è sufficiente prendere dall'ambiente il suo valore.

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Den id          -> lookup env id
```

Operazioni di base

Per valutare le operazioni di base usiamo delle funzioni ausiliarie.

```
let int_add (n : evT) (m : evT) : evT =
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a + b)
  | _, _, _, _               -> raise TypeError

let int_sub (n : evT) (m : evT) : evT =
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a - b)
  | _, _, _, _               -> raise TypeError

let int_times (n : evT) (m : evT) : evT =
```

```
  match typecheck "int" n, typecheck "int" m, n, m with
  | true, true, Int a, Int b -> Int (a * b)
  | _, _, _, _               -> raise TypeError

let is_zero (n : evT) : evT =
  match typecheck "int" n, n with
  | true, Int a -> Bool (a = 0)
  | _, _        -> raise TypeError

let int_equals (v1 : evT) (v2 : evT) : evT =
  match typecheck "int" v1, typecheck "int" v2, v1, v2 with
  | true, true, Int n, Int m -> Bool (n = m)
  | _, _, _, _               -> raise TypeError

let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Add (e1, e2)    -> int_add (eval e1 env) (eval e2 env)
  | Sub (e1, e2)    -> int_sub (eval e1 env) (eval e2 env)
  | Times (e1, e2)  -> int_times (eval e1 env) (eval e2 env)
  | IsZero e        -> is_zero (eval e env)
  | Equals (e1, e2) -> int_equals (eval e1 env) (eval e2 env)
```

Costrutto condizionale

Per valutare il costrutto condizionale ci rifacciamo alle regole di semantica date nelle sezioni precedenti: in particolare le regole operazionali per l'`if-then-else` sono

$$\frac{\texttt{env} \triangleright \texttt{cond} \Rightarrow \texttt{True} \quad \texttt{env} \triangleright e_1 \Rightarrow v_1}{\texttt{env} \triangleright \texttt{IfThenElse}(\texttt{cond}, e_1, e_2) \Rightarrow v_1}$$

$$\frac{\texttt{env} \triangleright \texttt{cond} \Rightarrow \texttt{False} \quad \texttt{env} \triangleright e_1 \Rightarrow v_1}{\texttt{env} \triangleright \texttt{IfThenElse}(\texttt{cond}, e_1, e_2) \Rightarrow v_2}$$

Possiamo quindi esprimere la regola in OCaml aggiungendo la seguente clausola:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | IfThenElse (cond, e1, e2) ->
    ( let evalCond = eval cond env in
        match typecheck "bool" evalCond, evalCond with
        | true, Bool true   -> eval e1 env
        | true, Bool false  -> eval e2 env
        | _                 -> raise TypeError )
```

Costrutto `let`

Anche per quanto riguarda il `let` ci rifacciamo alla regola operazionale:

$$\frac{\texttt{env} \triangleright e_1 \Rightarrow v' \quad \texttt{env}[x := v'] \triangleright e_2 \Rightarrow v}{\texttt{env} \triangleright \texttt{Let}(x, e_1, e_2) \Rightarrow v}$$

da cui la corrispondente regola dell'interprete è

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
```

```
  | Let (id, e1, e2) ->
    let newEnv = (bind env id (eval e1 env)) in
      eval e2 newEnv
```

Astrazione funzionale

Vogliamo ora aggiungere al linguaggio un meccanismo per costruire funzioni anonime. Per semplicità consideriamo funzioni con un singolo parametro, dato che grazie al *currying* possiamo ottenere gratuitamente le funzioni con più parametri.

Come descritto nella sintassi astratta, il costruttore di una funzione anonima è `Fun of ide * expr`. I due parametri richiesti sono

- il nome del parametro della funzione;
- il corpo della funzione, che è un'altra espressione.

Come abbiamo visto precedentemente nel caso di linguaggi funzionali con scoping statico valutare una funzione equivale ad ottenere una *chiusura*. Aggiungiamo quindi il tipo delle chiusure ai tipi esprimibili:

```
type evT =
  | Int     of int
  | Bool    of bool
  | Closure of ide * expr * evT env
  | Unbound
```

Una chiusura è quindi data da

- il nome del parametro formale (di tipo `ide`);
- il codice della funzione dichiarata (di tipo `expr`);
- l'ambiente di dichiarazione della funzione (di tipo `evT env`).

La regola operativa è quindi ovvia:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Fun (id, body) -> Closure (id, body, env)
```

Applicazione di funzione

Avere la possibilità di dichiarare funzioni senza poterle valutare è inutile, quindi sfruttiamo il costrutto `Apply` per applicare le funzioni dichiarate con il costrutto `Fun`.

La regola per l'applicazione di funzione è la seguente:

$$\frac{\texttt{env} \triangleright \texttt{f} \Rightarrow \texttt{Closure}(\texttt{id}, \texttt{body}, \texttt{funDeclEnv}) \qquad \texttt{env} \triangleright \texttt{arg} \Rightarrow \nu_{\texttt{arg}} \qquad \texttt{funDeclEnv}[\texttt{id} := \nu_{\texttt{arg}}] \triangleright \texttt{body} \Rightarrow \nu}{\texttt{env} \triangleright \texttt{Apply}(\texttt{f}, \texttt{arg}) \Rightarrow \nu}$$

Analizziamo un momento cosa ci dice la regola: supponiamo di voler valutare in un ambiente `env` l'espressione `Apply(f, arg)`. Allora dobbiamo

(1) valutare `f` e verificare che sia una chiusura, formata da

- `id`: il nome del parametro formale,

- `body`: il corpo della funzione,
- `funDeclEnv`: l'ambiente di dichiarazione della funzione;

(2) valutare il parametro attuale `arg`, ottenendo un valore $v_{\texttt{arg}}$;

(3) valutare nell'ambiente di dichiarazione della funzione arricchito dal legame tra il parametro formale `id` e il valore del parametro attuale $v_{\texttt{arg}}$ il corpo della funzione (ovvero `body`).

A questo punto è semplice tradurre la regola in codice OCaml:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Apply(f, arg) ->
    ( match eval f env with
      | Closure(id, body, funDeclEnv) ->
        let valArg = eval arg env in
        let newEnv = bind funDeclEnv id valArg in
          eval body newEnv
      | _ -> failwith "not a functional value" )
```

Funzioni ricorsive

L'interprete definito finora non ci consente di dichiarare o valutare funzioni ricorsive: infatti come abbiamo visto precedentemente nelle chiusure ricorsive deve essere presente anche la funzione stessa. Per poter dichiarare funzioni ricorsive dobbiamo quindi estendere la sintassi del nostro linguaggio con un nuovo costrutto:

```
type expr =
  | ...
  | LetRec of ide * ide * expr * expr
```

Una dichiarazione di funzione ricorsiva è quindi una versione particolare del costrutto `let` formata da

- il nome della funzione,
- il nome del parametro,
- il corpo della funzione,
- l'espressione in cui applichiamo la funzione ricorsiva.

Abbiamo anche bisogno di estendere i tipi esprimibili per includere le chiusure ricorsive:

```
type evT =
  | ...
  | RecClosure of ide * ide * expr * evT env
```

In questo caso i parametri sono sempre nome della funzione e del parametro, corpo della funzione e ambiente al momento della dichiarazione della funzione.

La regola operazionale per valutare un `let rec` è equivalente a quella del `let`, soltanto che dobbiamo costruire la chiusura ricorsiva prima di valutare l'espressione principale.

$$\frac{\texttt{env}[\texttt{f} := \texttt{r}] \triangleright \texttt{letBody} \Rightarrow v \qquad \text{dove } \texttt{r} = \texttt{RecClosure}(\texttt{f}, \texttt{id}, \texttt{funBody}, \texttt{env})}{\texttt{env} \triangleright \texttt{LetRec}(\texttt{f}, \texttt{id}, \texttt{funBody}, \texttt{letBody}) \Rightarrow v}$$

La corrispondente regola in OCaml diventa

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | LetRec(f, id, funBody, letBody) ->
    let r = RecClosure(f, id, funBody, env) in
    let newEnv = bind env f r in
      eval letBody newEnv
```

Anche in questo caso dobbiamo permettere l'applicazione di una funzione ricorsiva ad un parametro: dobbiamo quindi espandere il costrutto `Apply` con una nuova regola operazionale.

$$\frac{\begin{array}{l}\texttt{env} \triangleright \texttt{f} \Longrightarrow \texttt{closure} \\ \texttt{env} \triangleright \texttt{arg} \Longrightarrow v_{\texttt{arg}} \\ \texttt{newEnv} \triangleright \texttt{body} \Longrightarrow v \\ \text{dove } \texttt{closure} = \texttt{RecClosure}(\texttt{funId}, \texttt{id}, \texttt{body}, \texttt{funDeclEnv}) \\ \texttt{newEnv} = \texttt{funDeclEnv}[\texttt{f} := \texttt{closure}, \texttt{id} := v_{\texttt{arg}}]\end{array}}{\texttt{env} \triangleright \texttt{Apply}(\texttt{f}, \texttt{arg}) \Longrightarrow v}$$

La regola ci dice che per valutare un'espressione della forma `Apply(f, arg)` dobbiamo

- valutare `f` e usare questa regola se il risultato è una chiusura ricorsiva;
- valutare il parametro formale `arg`, ottenendo il valore $v_{\texttt{arg}}$;
- calcolare l'ambiente in cui valuteremo il corpo della funzione, che è l'ambiente di dichiarazione (`funDeclEnv`) insieme al legame tra la funzione `f` e la sua chiusura e a quello tra il nome del parametro formale (`id`) e il valore del parametro attuale ($v_{\texttt{arg}}$);
- valutare il corpo della funzione in questo nuovo ambiente.

Possiamo quindi finalmente ampliare l'interprete:

```
let rec eval (e : expr) (env : evT env) : evT =
  match e with
  | ...
  | Apply(f, arg) ->
    ( match eval f env with
      | Closure(id, body, funDeclEnv) -> ...
      | RecClosure(funId, id, body, funDeclEnv) as funClosure ->
        let actualVal = eval arg env in
        (* bind between f and the closure *)
        let recEnv = bind funDeclEnv funName funClosure in
        (* bind between the formal parameter and the actual value *)
        let actualEnv = bind recEnv param actualVal in
          eval funBody actualEnv
      | _ -> failwith "not a functional value" )
```

5

Semantica dei linguaggi Object-Oriented

5.1 CLASSI E METODI

Vogliamo quindi estendere lo studio dei blocchi e delle astrazioni procedurali in generale allo studio di classi e oggetti. La differenza principale con quanto studiato prima è che le classi ci forniscono un meccanismo per creare un ambiente e una memoria *permanenti*, cioè che sopravvivono alla creazione dell'oggetto, e *accessibili* e *utilizzabili* da chiunque possieda il loro meccanismo di accesso.

Sappiamo che un tale meccanismo di astrazione può essere utilizzato nella maggior parte dei linguaggi orientati agli oggetti attraverso la parola chiave **new**: essa viene usata per *attivare* un nuovo oggetto e con esso il suo ambiente locale.

Implementazione delle variabili di istanza

Nell'ambiente locale di un oggetto sono certamente contenute le sue variabili di istanza. La strategia che utilizziamo per implementare le variabili di istanza è un semplice ambiente locale statico contenente i legami tra i nomi delle variabili (e il loro descrittore di tipo) e i loro valori. Ad esempio se considerassimo il codice

```
class A {
    int a1;
    int a2;
}
A obj = new A(4, 5);
```

l'ambiente locale statico di `obj` è formato da due campi (uno per ogni variabile di istanza) che contengono i due binding, più dello spazio per il descrittore di tipo della classe `A`.

Tuttavia sappiamo che il meccanismo dell'ereditarietà ci consente di avere variabili non dichiarate esplicitamente in una classe, ma ereditate dalla classe padre, come nel seguente esempio:

```
class A {
    int a1;
    int a2;
}
class B extends A {
    int b;
}
```

Ogni oggetto di tipo `B` conterrà all'interno del suo ambiente locale (e in particolare all'inizio) i campi relativi alle variabili ereditate.

In questo modo risolviamo molto semplicemente alcuni possibili problemi:

- la gestione dell'ereditarietà singola è immediata;
- la gestione dello *shadowing* (la sottoclasse ha variabili con lo stesso nome della classe padre) è altrettanto immediata;
- se sono previsti meccanismi di controllo statico è facile implementare un accesso diretto ai vari campi dell'oggetto tramite un *indirizzo di base* e un *offset*.

Metodi e dispatching

I metodi di una classe sono comuni a tutti gli oggetti di una data classe, quindi viene naturale memorizzarli in un'unica area di memoria a cui si può accedere conoscendo il tipo della variabile che stiamo considerando.

Tuttavia, l'implementazione dei metodi non può essere così immediata. Infatti consideriamo il caso delle *interfacce* in Java:

```
List<String> l = ...
l.add("ciao");
```

Essendo `List` solo un'interfaccia, essa non implementa direttamente il metodo `add`: esso viene implementato dalle varie classi che rispettano l'interfaccia `List` in modi diversi, quindi per eseguirlo dobbiamo conoscere la classe dell'oggetto `l`. L'invocazione di un metodo deve quindi necessariamente *passare per l'oggetto*: questa nozione viene chiamata **dynamic dispatch**.

La soluzione al problema delle interfacce è quindi quello di associare alla classe una tabella (detta **tabella dei metodi** oppure **dispatch table/vector**) che contiene il binding dei metodi e il descrittore di tipo della classe: ogni oggetto della classe contiene un riferimento alla tabella dei metodi, dunque possiamo eseguire ogni metodo della classe anche quando lavoriamo con le interfacce.

I metodi in sé sono implementati come normali funzioni, quindi quando sono invocati creano un record di attivazione sullo stack con i campi già visti in precedenza. Tuttavia ogni metodo deve poter accedere alle variabili di istanza dell'oggetto: l'oggetto diventa quindi un *parametro implicito* del metodo, e il riferimento ad esso è dato dal puntatore `this`.

Come risolviamo i problemi di ereditarietà per quanto riguarda i metodi? Esistono due soluzioni.

Soluzione Smalltalk

Nel linguaggio Smalltalk i dispatch vector di classi in gerarchia formano una *lista di tabelle*: la tabella dei metodi di una classe contiene solo i metodi effettivamente implementati dalla classe; per accedere ai metodi della classe padre bisogna risalire la lista di tabelle fino a trovare il metodo cercato. Questa soluzione è particolarmente semplice, ma porta ad un discreto overhead in run-time.

Sharing Strutturale

La soluzione di **sharing strutturale** è quella adottata da linguaggi come il C++ e il Java: ogni classe ha la sua tabella dei metodi contenente i nomi dei metodi e dei *puntatori* al codice. In questo modo una classe che eredita un metodo deve semplicemente inizializzare il puntatore al codice già creato precedentemente. In particolare ogni metodo è identificabile staticamente attraverso il suo offset nella tabella dei metodi, quindi l'overhead a runtime della prima soluzione è azzerato.

Metodi statici

I metodi statici vengono implementati come se fossero delle procedure dei linguaggi imperativi: non potendo accedere alle variabili di istanza non hanno bisogno del puntatore `this` all'oggetto, e quindi sono semplicemente delle funzioni nell'ambiente globale.

Compilazione separata

Alcuni linguaggi (come il Java) implementano il meccanismo della *compilazione separata delle classi*: la compilazione di una classe produce del codice che viene caricato dalla macchina astratta del linguaggio *dinamicamente* quando il programma effettua un riferimento alla classe (**class loading**).

In questo caso gli offset non possono essere calcolati staticamente in quanto si possono fare modifiche non visibili alla struttura delle classi.

5.2 JAVA VIRTUAL MACHINE

Studiamo ora i meccanismi di funzionamento del Java ed in particolare la Java Virtual Machine (JVM).

Un'applicazione Java viene eseguita da un programma chiamato **Java Run-Time**, contenente la JVM e la Java Class Library (JCL), che contiene tutte le librerie del Java.

La Java Virtual Machine è la macchina astratta del Java: essa descrive il *class file format*, ovvero il formato e i vincoli (strutturali e sintattici) che devono essere rispettati da tutti i file `.class` a prescindere dalla macchina su cui vengono compilati.

I file prodotti dal compilatore `.javac` non sono quindi specifici per un solo tipo di macchina, ma sono universali: è la JVM che si occupa di compilarli nel linguaggio **bytecode**, che è il linguaggio macchina usato dalla JVM. In particolare quindi la JVM è formata da

- il *loader*, usato per caricare i file;
- il *verifier*, usato per controllare che i file `.class` caricati rispettino le specifiche e non siano quindi malevoli/malformati;
- in *linker*, che collega le librerie e i vari file tra loro, restituendo il bytecode;
- il *bytecode interpreter*, che esegue il programma.

Perché il doppio livello di compilazione?

- Innanzitutto perché la specifica della JVM non prescrive come deve essere implementata, quindi in questo modo possono essere implementate forme diverse della macchina virtuale.
- Inoltre le macchine hardware non possono avere a che fare con alberi di sintassi astratta, ma operano direttamente sui registri: astraendo sulla macchina hardware possiamo generare un linguaggio vicino alla macchina che però opera su strutture superiori all'architettura.

In particolare il bytecode Java sfrutta uno stack invece dei registri: le operazioni sono fatte sullo stack e lo modificano in modo incrementale, nel senso che ogni operazione viene eseguita sugli elementi che si trovano in testa allo stack e il risultato viene messo in testa allo stack. Lo stack degli operandi viene utilizzato per

- trasmettere i parametri ai metodi;
- restituire il risultato di un metodo;
- memorizzare i risultati intermedi delle operazioni;
- memorizzare le variabili locali.

La JVM può quindi essere descritta come una macchina astratta, stack-based e multi-threaded. In particolare ogni thread della JVM contiene

- un **program counter**, contenente l'indirizzo dell'istruzione corrente del metodo in esecuzione;
- uno stack chiamato **ambiente**, contenente i record di attivazione (chiamati **frame**) dei metodi. Ogni frame contiene
  - un array con le variabili locali (**local variable array**);
  - uno spazio per il valore di ritorno;
  - lo stack degli operandi;
  - un riferimento alle informazioni di runtime, contenute nella **constant pool**.

In particolare il local variable array contiene tutte le variabili locali del metodo e tutti i parametri, incluso `this` nel caso di metodi non statici. Nel caso di metodi non statici quindi la posizione 0 è riservata al puntatore `this`, mentre nel caso di metodi statici le variabili locali e i parametri partono dall'indice 0.

Ogni frame inoltre contiene un puntatore ad una struttura particolare, denominata **constant pool**: essa contiene il descrittore di tipo associato alla classe dove è definito il metodo in esecuzione, e viene usato per fare il linking dinamico. Infatti quando una classe Java viene compilata, tutti i riferimenti a variabili e metodi nella classe sono memorizzati nella constant pool come riferimenti simbolici (nel senso che non indicano effettivamente una locazione fisica di memoria, ma solo logica).

I riferimenti simbolici così introdotti possono essere risolti con le seguenti strategie:

- possono essere risolti quando le classi vengono caricate, con una strategia *eager*;
- possono essere risolti la prima volta che durante l'esecuzione si incontra il riferimento, con una strategia chiamata *lazy resolution*; in particolare se il riferimento è per una classe che ancora non è stata caricata, il programma si occupa di caricarla con una strategia lazy (**lazy class loading**);
- ogni riferimento risolto diventa un offset rispetto alla struttura di memorizzazione a runtime.

Lo spazio dedicato alle classi nella JVM viene chiamato **Class Area**. Per ogni classe essa contiene:

- riferimento al classloader;
- la constant pool;
- i dati relativi alle variabili di istanza;
- i dati relativi ai metodi;
- il codice dei metodi.

Il bytecode generato dal compilatore `javac` viene memorizzato in un file `.class` contenente il bytecode dei metodi della classe e la constant pool della classe: quest'ultima viene usata nel class loading per risolvere i riferimenti simbolici nominati precedentemente.

Gli offset di accesso ai metodi non possono invece essere determinati staticamente: essi vengono determinati dinamicamente la prima volta che si trova un riferimento all'oggetto, mentre per eventuali accessi successivi si utilizza l'offset già calcolato.

5.3 GARBAGE COLLECTION

Abbiamo visto studiando la JVM che durante l'esecuzione di un programma Java si usano tre tipi di memoria:

- una *static area* dove vengono allocati contenuti definiti a tempo di compilazione;
- un *run-time stack*, che è di dimensione variabile e viene usato per allocare i vari record di attivazione dei sottoprogrammi e dei blocchi;
- uno *heap*, di dimensione fissa o variabile, usato per allocare oggetti e strutture dati dinamiche tramite il costrutto `new`.

Nella prima area vi sono generalmente le variabili globali, le variabili locali dei sottoprogrammi (non ricorsivi), le costanti determinate a tempo di compilazione, le tabelle usate dal supporto a runtime per effettuare ad esempio il type checking e in generale tutte le entità che hanno un indirizzo di memoria assoluto che viene mantenuto per tutta l'esecuzione del programma.

Nello stack, come abbiamo visto diverse volte, abbiamo un record di attivazione per ogni istanza di un sottoprogramma con le informazioni relative a quella determinata istanza.

Lo **heap** invece è la regione di memoria in cui possiamo allocare e deallocare blocchi di memoria in momenti arbitrari. Esso è necessario quando il linguaggio permette

- allocazione esplicita di memoria a run-time (come con il costrutto `new` o con la primitiva `malloc` del C);
- oggetti di dimensioni variabili;
- oggetti con vita non esprimibile da una struttura dati LIFO come una pila.

Tuttavia una corretta gestione dello heap non è banale: vediamo innanzitutto delle possibili implementazioni.

Implementazione con blocchi di dimensione fissa

La prima implementazione dello heap può essere la seguente: scegliamo un'area di memoria da usare come heap e suddividiamola in blocchi di dimensione fissa (abbastanza limitata). Questi blocchi vengono usati per formare una lista (detta *lista libera*) contenente tutti i blocchi che non sono stati allocati dal programma.

Inizialmente la lista contiene tutti i blocchi dello heap, ma ogni volta che viene allocato un oggetto tramite la `new` alcuni di questi blocchi vengono rimossi dalla lista libera e usati per contenere le informazioni dell'oggetto creato. Quando la memoria occupata dall'oggetto viene liberata i corrispondenti blocchi vengono restituiti alla lista libera e possono essere riusati per successive allocazioni.

Questo metodo è semplice, ma porta ad una grande frammentazione dello heap poiché la dimensione fissa dei blocchi non è necessariamente la più adatta in ogni circostanza.

Heap con blocchi variabili

In questa seconda implementazione inizialmente lo heap è formato da un unico blocco. Ad ogni richiesta di allocazione si cerca il blocco di dimensione opportuna usando una delle seguenti due strategie:

- **first fit**: si sceglie il primo blocco grande abbastanza per contenere l'oggetto da allocare;
- **best fit**: tra i blocchi grandi a sufficienza per contenere l'oggetto da allocare si sceglie il blocco più piccolo (con il minor spreco di spazio quindi).

Se il blocco scelto è molto più grande del necessario viene diviso in due e la parte inutilizzata viene restituita alla lista libera; stessa cosa succede quando il contenuto di un blocco viene deallocato, ed in particolare se un blocco adiacente è libero i due vengono fusi in un unico blocco.

Garbage collection

In alcuni linguaggi (come ad esempio il C/C++) la gestione della memoria viene lasciata interamente al programmatore: ogni blocco di memoria allocato dinamicamente va deallocato dinamicamente. Seppure questa scelta garantisca la massima flessibilità ed efficienza, un uso scorretto della memoria fa emergere diversi problemi, come ad esempio

- memory leaks, che avvengono quando strutture vengono allocate e mai deallocate;
- dangling reference, quando dei puntatori continuano a puntare ad aree di memoria deallocate;
- heap fragmentation, che si ha quando la lista libera dello heap è molto frammentata a seguito di diverse allocazioni e deallocazioni.

In più, da un punto di vista più concettuale, la gestione esplicita della memoria viola il principio di astrazione: è necessario quindi che questo processo sia automatizzato.

Il **Garbage Collector** è un componente della macchina virtuale che esegue un programma di un dato linguaggio: ad esempio linguaggi come Lisp, Scheme, Java, Haskell contengono un garbage collector nella loro macchina virtuale, mentre le macchine virtuali del C e C++ generalmente no.

Vediamo i principali problemi da risolvere.

Frammentazione

Esistono due tipi di frammentazione: la **frammentazione interna** e la **frammentazione esterna**.

Si parla di **frammentazione interna** quando viene allocato un blocco di dimensione maggiore di quella strettamente necessaria: una parte dello spazio allocato è sprecato.

Si parla invece di **frammentazione esterna** quando vogliamo allocare un blocco di una certa dimensione x e la quantità di memoria libera ce lo consente, ma la memoria libera è frammentata e ognuno dei blocchi liberi non è abbastanza grande per allocare un blocco di dimensione x.

Gestione dell'allocazione

Dobbiamo inoltre decidere quale tipo di strategia usare per allocare i blocchi: la strategia first fit è più veloce ma spreca più memoria, mentre la strategia best fit è più lenta ma più efficiente nella gestione della memoria.

Identificazione dei blocchi da deallocare

Un'altra scelta che dobbiamo fare è come decidere quali blocchi vanno deallocati. Nel caso di una deallocazione esplicita è semplice, in quanto basta seguire il puntatore passato come parametro alla funzione `free` (nel caso del C): tuttavia questo meccanismo può causare i problemi visti precedentemente e quindi si preferisce avere una gestione automatica.

Nel caso di deallocazione automatica diciamo che una porzione di memoria è recuperabile (nel senso che può essere deallocata) se non è più *raggiungibile* dal resto del programma. Dobbiamo quindi capire come definire precisamente il concetto di raggiungibilità.

Tecniche di garbage collection

Studiamo ora le principali tecniche di garbage collection.