

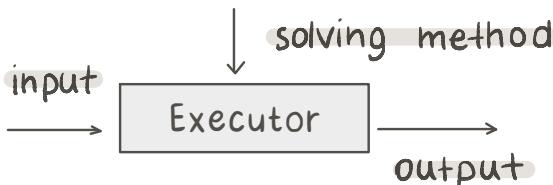
- Esistono problemi irrisolvibili?

Si esistono problemi irrisolvibili. Esistono anche problemi che possono essere risolti solo da determinati linguaggi.

Algoritmo = sequenza finita di passi che risolve una classe di problemi in un certo tempo finito.

Un programma può anche non terminare mai (Sistema Operativo).

Esecuzione



Esistono vari esecutori:

- Approccio astratto (macchine a stati finiti)
- Approccio funzionale (funzioni matematiche)
- Approccio della riscrittura.

NB: Se un problema non può essere risolto da una macchina super potente allora non è risolvibile.

MACCHINE DI BASE

I → input

O → output

Mfn → I → O machine function

Esempio di un bool:

I → {{0,1} × {0,1}}

O → {0,1}

Mfn → AND logico

Limiti : - Dobbiamo definire tutte le possibili configurazioni dell'input ;
- Non si può salvare lo stato della macchina

Non c'è memoria , non posso fare il riconoscimento di una stringa oppure fare la somma di un a sequenza di numeri. O

MACCHINE A STATI FINITI

I → input

O → output

S → stati

Mfn → I × S → O machine function

Sfn → I × S → S status function

Limiti : - Hanno una memoria finita ed il risultato dipende dall'input e dallo stato

dobbiamo a priori la dimensione dell'input

MACCHINE DI TURING

I → input
O → output
S → stati
Mfn → I × S → O machine function
Sfn → I × S → S status function
Dfn → I × S → D,D direzione testina (left,right,none)

Esempio palindrome :

$$A = \{0, 1, \wedge, : , :()\}$$
$$S = \{\text{HALT}, s_0, s_1, s_2, s_3, s_4, s_5\}$$

La macchina di Turing è la macchina più potente che conosciamo, se infatti un problema non è risolvibile da questa macchina viene considerato irrisolvibile.

Problemi irrisolvibili ?

Se la macchina a cui diamo in pasto il problema è programmata correttamente e se il problema è risolvibile, la macchina ritorna un risultato corretto.

Se invece non riesce a ritornare un risultato corretto la macchina non si ferma, entriamo in un loop infinito.

Ma cosa prendono in input le macchine di Turing? Problemi?

- Le macchine di Turing computano funzioni NON problemi, dobbiamo quindi trovare il modo di esprimere i problemi in termini di funzioni.

└

X → gruppo di input

Y → gruppo di output

Funzione caratteristica $f(p)$ che lega ad ogni input $x \in X$ un output $y \in Y$

$$f(p) : X \rightarrow Y$$

- se $f(p)$ è computabile dalla macchina allora il problema è RISOLVIBILE
- se $f(p)$ non è computabile dalla macchina allora il problema è IRRISOLVIBILE

La domanda sorge quindi spontanea, quando una funzione è COMPUTABILE?

- Una funzione $f(p)$ si dice computabile se esiste una macchina di Turing che è in grado di:

– data la rappresentazione dell'input dopo un numero finito di passi produrre un output corretto.

└

f non è computabile se la macchina di Turing non può produrre un output corretto in un numero finito di passi (loop infinito).

Esempio di funzione definibile ma non computabile

Il problema dell' HALT

Questo problema può essere definito ma NON computato.

DATI:

- $M \rightarrow$ insieme di tutte le macchine di Turing
 - |
 - $m \in M \rightarrow$ una generica macchina di Turing
- $X \rightarrow$ un generico insieme di input
 - |
 - $x \in X \rightarrow$ un generico input
- $*$ \rightarrow un risultato indefinito, quando non riesce a darci una risposta

$$f_H(m, x) = \begin{cases} 1, & \text{se } m \text{ con input } x \text{ si ferma (risponde)} \\ 0, & \text{se } m \text{ con input } x \text{ non si ferma (non risponde)} \end{cases}$$

Processo di Gödel: ad ogni macchina di Turing può essere associato un numero naturale

Consideriamo un'altra funzione $g(\text{HALT})$ a cui passiamo il numero naturale associato come descrizione (quindi passiamo se stessa come input)

$$g_{\text{HALT}}(n_g) = \begin{cases} 1, & \text{se } f_H(n_g, n_g) = 0 \\ *, & \text{se } f_H(n_g, n_g) = 1 \end{cases}$$

Questo è assurdo perché stiamo dicendo che:

- se f_H si ferma allora $g(\text{HALT})$ ritorna che non si ferma;
- se f_H non si ferma allora $g(\text{HALT})$ ritorna che si ferma;

Decidibilità di un gruppo

- Dato un elemento dobbiamo capire se tale appartiene al gruppo oppure no.

Gruppo numerabile \rightarrow gruppo i cui elementi possono essere numerati, le cui funzioni potrebbero essere non computabili.

Gruppo numerabile ricorsivamente \rightarrow un gruppo numerabile le cui funzioni sono sempre computabili, possiamo cioè generare un elemento ma non significa che quell'elemento appartiene all'insieme

|

- esempio numero primo :

Chiediamo alla MT di generare tutti i numeri primi, adesso se in input diamo un insieme relativamente piccolo 100 ad esempio allora la macchina calcolerà tutti i numeri primi fino a 100, ma se aumentiamo la dimensione dell'input e la facciamo tendere ad infinito allora non sappiamo le macchina non ci da come output il numero che vogliamo perché non appartiene all'insieme o perché ancora non è arrivato a calcolarlo.

Gruppo decidibile → un gruppo i cui elementi possono essere numerati ed esiste un algoritmo che finisce dopo un numero finito di passi e sa decidere se un elemento appartiene al gruppo.



teoremi dei gruppi decidibili:

- 1° → un gruppo decidibile è anche semi-decidibile
- 2° → un gruppo G è decidibile solo se N-G è semidecidibile

Siamo interessati ai gruppi decidibili perché un linguaggio è un gruppo di parole ed i linguaggi di programmazione sono gruppi di parole (alfabeti) che seguono una rigida grammatica

Grammatica → notazione formale attraverso la quale specifichiamo la sintassi di un linguaggio.



VT : gruppo finito di simboli terminali (caratteri o stringhe di alfabeto A)

VN : gruppo finito di simboli non terminali (categorie sintattiche)

L'unione di VT e VN forma un vocabolario della grammatica

Una forma sentenziale è una stringa che contiene simboli terminali e non.

Una parola è una forma sentenziale composta da soli simboli terminali.

Esempio :

SUBJECT VERB PREDICATE



Non-terminali

bob washes the car



Terminali

GRAMMATICA FORMALE

Una **Grammatica** è una **notazione formale** con cui esprimere **in modo rigoroso** la **sintassi** di un linguaggio.

Una grammatica è una quadrupla $\langle VT, VN, P, S \rangle$ dove:

- **VT** è un **insieme finito di simboli terminali**
- **VN** è un **insieme finito di simboli non terminali**
- **P** è un **insieme finito di produzioni, ossia di regole di riscrittura** $\alpha \rightarrow \beta$ dove α e β sono stringhe: $\alpha \in V^*$, $\beta \in V^*$
 - ogni regola esprime una trasformazione lecita che permette di scrivere, nel contesto di una frase data, una stringa β al posto di un'altra stringa α .
- **S** è un particolare simbolo non-terminali detto **simbolo iniziale o scopo della grammatica**

Esempio : EXP=EXP OP EXP
 EXP=CONST
 CONST = (1 2 3 4 5 6 7 8 9 10)
 OP = (+ - * /)

DERIVAZIONE

Siano α, β due stringhe $\in (VN \cup VT)^*$, $\alpha \neq \epsilon$

Si dice che β deriva direttamente da α ($\alpha \rightarrow \beta$) se

- le stringhe α, β si possono decomporre in
 $\alpha = \eta A \delta$ $\beta = \eta \gamma \delta$
 ed esiste la produzione $A \rightarrow \gamma$.

$$\eta A \delta$$

Si dice che β deriva da α (anche non direttamente) se

- esiste una sequenza di N derivazioni dirette che da α possono infine produrre β

$$\alpha = \alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \dots \rightarrow \alpha_N = \beta$$

Voglio esprimere 34+18

S=EXP
 EXP=EXP OP EXP
 EXP=CONST OP CONST
 CONST OP CONST
 3CONST OP CONST
 34CONST OP CONST
 34 OP CONST
 34 + CONST
 34 + 1CONST
 34 + 18 CONST
 34+18

GRAMMATICA DI TIPO 0

CLASSIFICAZIONE DI CHOMSKY TIPO 0

Le grammatiche sono classificate in 4 tipi
in base alla struttura delle produzioni

- **Tipo 0:**
nessuna restrizione sulle produzioni

In particolare, le regole possono specificare riscritture che accorcianno la forma di frase corrente.

Esempio (grammatica di tipo 0)

$$\begin{array}{llll} S \rightarrow aSBC & CB \rightarrow BC & SB \rightarrow bF & FB \rightarrow bF \\ FC \rightarrow cG & GC \rightarrow cG & G \rightarrow \epsilon & \end{array}$$

Possibile derivazione $S \rightarrow aSBC \rightarrow abFC \rightarrow abcG \rightarrow abc$
 $lung=4 \quad lung=3$

GRAMMATICA DI TIPO 1

CLASSIFICAZIONE DI CHOMSKY TIPO 1

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 1 (dipendenti dal contesto):**
 produzioni vincolate alla forma:

$$\beta A \delta \rightarrow \beta \alpha \delta$$

con $\beta, \delta \in V^*, \alpha \in V^+, A \in VN$

$$\alpha \neq \epsilon$$

Quindi, A può essere sostituita da α solo nel contesto $\beta A \delta$.
 Le riscritture non accorcianno ma la forma di frase corrente.

Esempio (grammatica di tipo 1)

$$\begin{array}{llll} S \rightarrow aBC | aSBC & & & \\ CB \rightarrow DB & DB \rightarrow DC & DC \rightarrow BC & \\ aB \rightarrow ab & bB \rightarrow bb & bC \rightarrow bc & cC \rightarrow cc \end{array}$$

Infatti, secondo la definizione $\beta A \delta \rightarrow \beta \alpha \delta$ si può trasformare un metasimbolo per volta (A), lasciando intatto ciò che gli sta intorno

Osserva: la lunghezza del lato destro delle produzioni non è mai inferiore a quella del lato sinistro.

ESEMPIO

$$\begin{array}{llll} S \rightarrow aBC | aSBC & \beta = \epsilon \quad \delta = \epsilon & & \\ CB \rightarrow DB & \beta = \epsilon \quad \delta = B & & \\ DB \rightarrow DC & \beta = D \quad \delta = \epsilon & & \\ DC \rightarrow BC & \beta = \epsilon \quad \delta = C & & \\ aB \rightarrow ab & \beta = a \quad \delta = \epsilon & & \\ bB \rightarrow bb & \beta = b \quad \delta = \epsilon & & \\ bC \rightarrow bc & \beta = b \quad \delta = \epsilon & & \\ cC \rightarrow cc & \beta = c \quad \delta = \epsilon & & \end{array}$$

GRAMMATICA DI TIPO 2

CLASSIFICAZIONE DI CHOMSKY TIPO 2

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 2: context free (*indipendenti dal contesto*):**

produzioni vincolate alla forma:

$$A \rightarrow \alpha$$

con $\alpha \in (VT \cup VN)^*$, $A \in VN$

Qui A può sempre essere sostituita da α , indipendentemente dal contesto.

Se α ha la forma u oppure $u B v$, con $u, v \in VT^*$ e $B \in VN$, la grammatica si dice *lineare*.

GRAMMATICA DI TIPO 3

CLASSIFICAZIONE DI CHOMSKY TIPO 3

Le grammatiche sono classificate in 4 tipi in base alla struttura delle produzioni

- **Tipo 3 (grammatiche regolari):** produzioni vincolate alle *forme lineari*:

lineare a destra

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

con $A, B \in VN$ e $\sigma \in VT^*$

lineare a sinistra

$$A \rightarrow \sigma$$

$$A \rightarrow B \sigma$$

Si intende che le produzioni di una data grammatica devono essere **TUTTE o lineari a destra, o lineari a sinistra – non mischiate**.

Si noti che σ può essere ϵ .

GRAMMATICHE REGOLARI CASO PARTICOLARE

Per grammatiche regolari è **sempre possibile e spesso conveniente trasformare la grammatica in forma strettamente lineare**

- non più $\sigma \in VT^*$ (σ è una stringa di caratteri)

lineare a destra lineare a sinistra

$$A \rightarrow \sigma$$

$$A \rightarrow \sigma B$$

$$A \rightarrow \sigma$$

$$A \rightarrow B \sigma$$

- bensì $a \in VT$ (a è un singolo carattere)

lineare a destra lineare a sinistra

$$X \rightarrow a$$

$$X \rightarrow a Y$$

$$X \rightarrow a$$

$$X \rightarrow Y a$$

GRAMMATICHE LINEARI: ESEMPI

$$VT = \{ a, +, - \}, VN = \{ S \}$$

- **Grammatica G1 (lineare a sinistra: $A \rightarrow B y$, con $y \in VT^*$)**

$$S \rightarrow a \quad S \rightarrow S + a \quad S \rightarrow S - a$$

- **Grammatica G2 (lineare a destra: $A \rightarrow x B$, con $x \in VT^*$)**

$$S \rightarrow a \quad S \rightarrow a + S \quad S \rightarrow a - S$$

- **Grammatica G3 (G2 resa strettamente lineare a destra)**

$$S \rightarrow a \quad S \rightarrow a A \quad A \rightarrow + S \quad A \rightarrow - S$$

- **Grammatica G4 (lineare a destra e anche a sinistra)**

$$S \rightarrow ciao$$

- **Grammatica G5 (G4 resa strettamente lineare a destra)**

$$S \rightarrow c T \quad T \rightarrow i U \quad U \rightarrow a V \quad V \rightarrow o$$

Paradigma Imperativo

- I linguaggi di programmazione imperativi sono scritti come una guida passo passo per il computer. Descrivono esplicitamente quali passaggi devono essere eseguiti ed in quale ordine per raggiungere la soluzione desiderata.
- I linguaggi di programmazione imperativa sono caratterizzati dal loro carattere istruttivo e pertanto di solito richiedono molte più righe di codice per ciò che può essere descritto come poche istruzioni nello stile dichiarativo.

Riassumendo la programmazione imperativa si concentra più sul "COME" che sul "COSA".

Paradigma Funzionale

Funzioni nei linguaggi imperativi

- Una funzione è solo un costrutto per encapsulare codice
- NON può essere assegnata ad una variabile (i puntatori a funzioni in C puntano al codice della funzione non all'entità funzione vera e propria)
- NON può essere passata come argomento di una funzione
- Una funzione non può essere creata e ritornata (solo i puntatori a funzioni esistenti possono essere ritornati)
- Sono entità definite staticamente

Funzioni nei linguaggi funzionali

- Una funzione è un tipo di dato che può essere maneggiato come gli altri tipi di dato (int, char ecc..)
- Può essere assegnata ad una variabile di tipo funzione
- Può essere ritornata da un'altra funzione(una funzione può essere creata all'interno di altre funzioni)
- Può essere creata e definita on the fly

Classi di una funzione

Innestare funzioni una dentro l'altra (passando una funzione come argomento di un'altra) porta a creare una gerarchia di funzioni e a definire quindi funzioni di ordine superiore.

- La funzione a cui è passato un a funzione come argomento sarà una funzione di secondo grado, una funzione di terzo grado sarà una funzione che ha come argomento una funzione e di secondo grado ecc..

Esempi di funzione in JavaScript

- Le funzioni sono entità di tipo <funzione>

```
var f1 = function (z) { return z*z; }
```

In questo caso la funzione definita non ha nome.

- Una funzione può ricevere un'altra funzione come argomento

```
var c = function (f, x) { return f(x); }
```

f parametro di tipo funzione

- Una funzione può ritornare il risultato di una sua funzione innestata

```
function createPlus10(x) {return  
function(r){return r+10}}
```

- Una funzione può essere creata testo

```
square = new Function("x", "return x*x")
```

Chiusure

Una chiusura è una funzione di prima classe definita all'interno di uno o più blocchi (sono quindi utilizzabili solo in linguaggi che ne fanno riferimento solo come entità di prima classe)

Hanno origine nei linguaggi funzionali (Lisp, Scheme..) ma recentemente sono disponibili nei linguaggi OO (Object Oriented)

- Il codice di una chiusura può far riferimento a variabili locali definite nei blocchi che la circondano
- Le variabili usate da una chiusura non sono deallocate fino a quando la chiusura è accessibile (i dati sono allocati nell'heap invece che sullo stack)

Esempi di chiusure in JavaScript

```
function ff(f, x) { // creates a closure  
    return function(r){ return f(x)+r; }  
}
```

- Ad ogni chiamata di ff viene creata una nuova funzione (senza nome) che ha bisogno di:
 - f, x, r per eseguire.
- Si verifica quindi la chiusura, che tiene in vita le variabili anche dopo che ff è stata eseguita.

```
var f1 = ff( Math.sin, 0.8)  
var f2 = ff( function(q){return q*q}, 0.8)
```

- f1 ed f2 contengono due funzioni differenti in quanto la prima calcolerà il seno di $0.8 + r$ e la seconda il quadrato di $0.8 + r$.

▶ Now we can call f1 and f2

```
var r1 = f1(3) // 3.717356091 [sin(0.8) + 3]
```

▶ Computes the sin of the 0.8 plus 3

```
var r2 = f2(0.36) // 1.0000000 [0.64 + 0.36]
```

▶ Computes the square of 0.8 plus 0.36

- Questo esempio mette in evidenza il funzionamento della chiusura in quanto x ed r sono sopravvissute anche alla fine della prima chiamata di ff.

Alcuni possibili usi della chiusura sono:

- nascondere le variabili della funzione, in modo che non siano accessibili dall'esterno.
- le varie funzioni di grado diverso possono utilizzare il loro stato per comunicare (una funzione figlia può accedere variabili della classe padre).

Lambda expression

Una chiusura è creata con una lambda expression, una funzione anonima (senza nome) che può essere usata direttamente nel corpo di un metodo.

Esempi di chiusure in vari linguaggi

JavaScript-> function() {body}

- Dato che JS è un linguaggio ad oggetti anche le funzioni sono oggetti, generate dal costruttore Function.

Scala-> (params) => body

- Come JS, anche scala è object-oriented

C++-> auto f = [ext_var_ref](args)

- Questa espressione crea una funzione lambda e quindi una chiusura.

C#-> { params => body }

Java8-> (params) -> {body}

- Come C++ e C# con l'unica differenza che qui le variabili sono read-only.

Alcuni esempi di utilizzo delle lambda expression in C++

- auto f = [=] (int x) accesses the external variables by copying their value
- auto f = [&] (int x) accesses the external variables by reference
- auto f = [&v, &u, &w] (int x) accesses v and w by reference, u by copy

► Some examples in C#

```
{ int x => x+1 }
{ int x, int y => x+y }

{ string s =>
    int.parse(s)
    { =>
        Console.WriteLine("hi")
    }
}
```

► Some examples in Java 8

```
(int x) -> x+1
(int x, int y) -> x+y

(String s) -> s + "a"
(int x) ->
{System.out.println(x);}
```

- Le variabili salvate in Java tramite chiusura sono read-only e devono essere necessariamente constanti (final), in caso contrario il compilatore si arrabbia.

```
import java.util.function.*;
public class ProvaChiusura {
    public static void main(String args[]) {
        int z = 23;
        IntFunction<Integer> h = (int x) ->
x + (z--);
        System.out.println(h.apply(2));
    }
}
```

```
1 error found:
File: F:\MyProg\java\Java8\ProvaChiusura.java [line: 6]
Error: local variables referenced from a lambda expression must be final or
effectively final
```

Chiusure dinamiche vs lessicali

Si può definire come e dove finisce la ricerca di variabili libere ?

- In presenza di funzioni innestate quale criterio prendiamo in considerazione per la valutazione delle variabili?

1) **Catena lessicale** -> il testo del programma contiene una catena di definizioni innestate, che seguono il testo del programma

2) **Catena dinamica** -> la chiamata delle funzioni a tempo di esecuzione crea una catena differente di record di attivazione

Dobbiamo quindi scegliere uno dei due criteri.

```
var x = 20;
function provaEnv(z) { return z + x; }
function testEnv() {
    var x = -1;
    return provaEnv(18);
}
```

- In questo esempio x fa riferimento alla sua definizione più vicina ($x=20$), la chiamata di funzione `provaEnv(18)` ritornerà infatti $18+20=38$

E' stato utilizzato un criterio di chiusura LESSICALE.

```
provaEnv
var x = 20;
function provaEnv(z) { return z + x; }
function testEnv() {
    var x = -1;
    return provaEnv(18);
}
```

- In questo esempio x fa riferimento alla sua definizione più RECENTE ($x=-1$) (l'ultima nello stack), la chiamata if funzione `provaEnv(18)` ritornerà infatti $18+(-1)=17$

E' stato utilizzato un criterio di chiusura DINAMICA.

PRO/CONTRO LESSICALE

- Molto leggibile, si può capire senza eseguire il programma :)
- Un simbolo è però legato staticamente ad una variabile :(

PRO/CONTRO DINAMICA

- Poco leggibile, si deve eseguire il codice mentalmente per capirla :(
- Non ha limiti statici :)

Siccome la leggibilità è molto importante molti linguaggi di programmazione **adottano la chiusura LESSICALE**.

Esempio in JavaScript

```
> var x = 20
< undefined
> function provaEnv(z) {return z+x}
< undefined
> function testEnv() {var x = -1; return provaEnv(18)}
< undefined
> testEnv()
< 38
```

Valutazione delle funzioni

Un linguaggio che utilizza le funzioni come entità di prima classe deve definire una strategia per la loro valutazione.

Strategia Applicativa

Una delle strategie più diffuse:

- **WHEN**: gli argomenti delle funzioni sono valutati subito
- **WHAT**: alla funzione viene passato il valore degli argomenti
- **HOW**: la funzione viene attivata dandole il controllo (modello sincrono)

Il modello applicativo è una valutazione call-by-value

PRO

- E' generale ed efficiente
- I valori passati sono semplici e facili da maneggiare
- Ottima leggibilità

CONTRO

- Se uno degli argomenti passato non serve alla funzione viene valutato comunque
- L'immediata valutazione di un argomento può causare errori

Esempio in Java di call-by-value

```
class Example {  
    static void printOne(boolean cond, double a, double b){  
        if (cond) System.out.println("result = " + a);  
        else System.out.println("result = " + b);  
    }  
    public static void main(String[] args) {  
        int x=5, y=4;  
        printOne(x>y, 3+1, 3/0); //ArithmetricException: / by zero  
    }  
}
```

- Nell'esempio utilizzando la strategia call-by-value tutti gli argomenti della funzione vengono valutati subito, ma il terzo elemento (3/0) non verrà mai utilizzato poiché la prima condizione è sempre vera(spreco di tempo).

Esempio in JavaScript di call-by-value

```
var f = function(flag, x) {  
    return (flag<10) ? 5 : x;  
}  
var b = f(3, abort()); // Error!!  
document.write("result =" + b);
```

- La funzione in questo caso si ferma subito a causa dell'immediata valutazione della funzione `abort()`, se non ci fosse stata una valutazione immediata degli argomenti la funzione `f` avrebbe funzionato correttamente poiché la condizione `3<10` è vera.

Valutazione Call-by-Name

- Gli argomenti non sono valutati quando la funzione è invocata, ma solo se e quando sono necessari.
- Gli argomenti passati alla funzione non sono né valori né indirizzi ma oggetti computabili che possono essere eseguiti.

Esempio di output se Java adottasse il call-by-name

```
class Example {  
    static void printOne(boolean cond, double a,  
double b){  
        if (cond) System.out.println("result = " + a);  
        else System.out.println("result = " + b);  
    }  
    public static void main(String[] args) {  
        int x=5, y=4;  
        printOne(x>y, 3+1, 3/0);  
    }  
}
```

- In questo esempio l'argomento (3/0) non viene mai valutato perché non è necessario alla funzione in quanto la condizione è vera

Esempio di output se JavaScript adottasse il call-by-name

```
var f = function(flag, x) {  
    return (flag<10) ? 5 : x;  
}  
var b = f(3, abort()); // Error!  
document.write("result =" + b);
```

- In questo esempio la funzione stampa 5 e non viene più fermata a priori a causa della valutazione precoce dell'argomento `abort()`.

Non molto utilizzato richiede troppe risorse a tempo di esecuzione poiché gli oggetti computabili passati come argomento sono più difficili da gestire.

Anche se questa strategia non è molto utilizzata può essere simulata nei linguaggi che usano le funzioni come entità di prima classe, un esempio:

► The original JavaScript example (applicative strategy):

```
var f = function(flag, x) { // two values  
    return (flag<10) ? 5 : x; // use the two values  
}
```

```
var b = f(3, abort());
```

```
document.write("result =" + b);
```

► The same example in a call-by-name fashion:

```
var f = function(flag, x) { // two functions  
    return (flag()<10) ? 5 : x(); // call the two  
functions  
}  
  
var b = f(function(){return 3}, function(){  
abort()});  
  
document.write("result =" + b);
```

► Now it works!

Il call-by-name può essere simulato anche in linguaggi che non trattano le funzioni come entità di prima classe, un esempio:

- ▶ The new code is:

```
class Example {  
    static void printTwo(boolean cond, MyFunction a,  
    MyFunction b) {  
        if (cond) System.out.println("result = " +  
        a.invoke() );  
        else System.out.println("result = " + b.invoke()  
    );  
    }  
    public static void main(String[] args) {  
        int x=5, y=4;  
        printTwo(x>y, new MyFunction1(), new  
        MyFunction2() );  
    }  
}
```

- ▶ It works!

- Questo esempio è mostrato attraverso l'utilizzo di Java7 in cui le funzioni non sono entità di prima classe, ma se proviamo a fare un confronto con lo stesso codice scritto in Java8, in cui le funzioni sono utilizzate come entità di prima classe, cosa succede? Il codice sarà più semplice?

```
static void printTwo(boolean cond, IntSupplier a,  
IntSupplier b){  
    if (cond) System.out.println("result = " +  
    a.getAsInt() );  
    else System.out.println("result = " + b.getAsInt()  
);  
}
```

- ▶ Not much easier than Java 7...

- In C# invece notiamo un miglioramento effettivo grazie all'utilizzo delle lambda expression e dei delegati

- ▶ Use of delegates in the example

```
class Example{  
    delegate double MyFunction();  
    static void printTwo(bool cond, MyFunction a,  
    MyFunction b){  
        if (cond)  
System.Console.WriteLine("result = " + a());  
        else System.Console.WriteLine("result =  
" + b());  
    }  
    public static void Main(string[] args) {  
        int x=5, y=4, a=0;  
        printTwo(x>y, () => 3+1, () => 3/a );  
    }  
}
```

- ▶ It works!

True function type

Can call as true functions

lambda expressions:
build functions

NB: a variable is needed to avoid the
compile-time computation of the
expression of constants (C# is efficient...)

LISP

Il Lisp è stato ideato nel 1958 da John McCarthy come linguaggio formale, per studiare le equazioni di ricorsione in un modello computazionale.

Il Lisp si presenta con un Prompt di inserimento comandi esattamente come farebbe una shell a comandi.

Inserendo una lista composta da elementi separati da spazi tra parentesi () otteniamo l'esecuzione, il processore Lisp valuta il comando inserito, sommando tutti termini che compongono la lista.

- Nel Lisp gli oggetti usati sono le S-expression (symbolic expressions) la cui sintassi è:
(comando liste liste liste ...)
- Il Lisp non tipizza il dato delle variabili.
- Il codice di un programma può essere scritto sottoforma di S-expression, utilizzando la notazione prefissa.

Il Lisp valuta il contenuto ed il tipo delle variabili solo quando una funzione esegue una operazione con il contenuto della S-expression.

The screenshot shows a terminal window titled 'C:\PROGRA~1\GCL-2.6.6-CLT\lib\gcl-2.6.6\unixport\saved_gcl.exe'. The window displays the following Lisp session:

```
Dedicated to the memory of W. Schelter
Use (help) to get some basic information on how to use GCL.

>(+ 5 7 8 9)
29
>(setq test 12)
12
>test
12
>(setq test 10)
10
>test
10
`
```

Variabili e Comandi basilari in Lisp

- Ogni **variabile** deve possedere un nome (sequenza alfanumerica di caratteri, non solo numeri) che la distingue dalla altre, in modo da poterla utilizzare semplicemente indicandone l'identificativo (nome).
- I dati immagazzinabili in una **variabile** sono :
 - numeri reali, numeri interi, stringhe di testo, liste di valori (cioè variabili contenenti più dati contemporaneamente)

- **Assegnamento:** (setq x 5) questa istruzione si limita ad inserire nella variabile di nome x il numero intero 5.

Le funzioni matematiche in Lisp

Lista di alcune funzioni disponibili per l'elaborazione numerica :

```
+ , - , * , / , 1+ , 1- , abs , gcd , min , max , zerop , atan , log , cos , sin , exp , sqrt ..
```

Esempio: (+ 10 7.5) esegue l'addizione tra i numeri 10 e 7.5

- La sintassi utilizzata: ([funzione] [valore1] [valore2]) è applicabile solamente alla quattro operazioni "+", "-", "*", "/".

Il Lisp utilizza per gli operatori aritmetici e binari la notazione infissa "polacca di Cambridge".

```
>(* (/ 7 6 ) (+ 4 (* 4 6)))  
98/3  
>(* (/ 1 2 ) 6)  
3  
>
```

Espressioni di confronto

- "**=**" valuta su due o più valori, passati come argomenti, sono uguali, in questo caso ritorna T altrimenti nil.
- "**/=**" valuta su due o più valori sono diversi, in questo caso ritorna T altrimenti nil.
- "**<**" e "**>**" valutano se un valore è minore o maggiore dell'elemento alla sua destra, in questo caso viene restituito T, altrimenti nil.
- **AND OR NOT NULL** usuali espressioni booleane(in particolare "NULL" ritorna T se il suo argomento è nil (falso) e nil se il suo argomento è T (vero).)

Istruzioni di controllo

Istruzione condizionale IF

```
(if [condizione]  
(progn [codice eseguito se condizione = vero] )  
(progn [codice eseguito se condizione = falso] ) )
```

- Il costrutto PROGN permette l'esecuzione di più linee di codice.
- Se non si utilizza PROGN viene eseguita una sola riga di codice per ognuno dei due stati.

Istruzioni condizionali

```
(cond <condizione1> <blocco1>
      <condizione2> <blocco2>
      ..
      .. )
```

Valutazione forzata

```
(eval <argomento>)
```

Funzioni

- Una funzione in Lisp così come in tutti gli altri linguaggi di programmazione non è altro che una parte di codice richiamabile attraverso l'utilizzo di un identificatore.

Sintassi:

```
(defun NomeFunzione (argomenti) (Corpo))
```

- Il valore che la funzione ritorna è il valore dell'ultima istruzione nel Corpo

Esempio 1: (defun stampamessaggio() (print "ciao"))

Esempio 2: (defun somma(x) (+ x x))

La funzione somma per x uguale a 3 si richiama ad esempio così: (somma 3)

```
>>(defun pow2 (a)
             (* a a))
POW2
>>(pow2 2)
4
>>(defun pow3 (a)
             (* a (pow2 a)))
POW3
>>(pow3 2)
8
>>
```

Assegnamento di variabili locali

```
let (<simbolo1> <valore1>).. (<simbolo2> <valore2>) <s-expression>)
```

Valori quotati

```
(quote <lista o atomi>)
```

Valutazione forzata

```
(eval <argomenti>)
```

► Examples

```
(quote (1 2 5))
```

```
→ (1 2 5)
```

```
'(1 2 5)
```

```
→ (1 2 5)
```

```
(setq e '(+ a 3))
```

```
→ (+ a 3)
```

```
e
```

```
→ (+ a 3)
```

► Examples

```
(setq a 5)
```

```
→ 5
```

```
(eval a)
```

```
→ 5
```

```
a
```

```
→ 5
```

Argomenti delle funzioni

► Arguments can be optional

► &optional

► (defun my-fun (m &optional n) ...)

► A function can have a variable number of arguments

► &rest

► (defun my-fun (k &rest l) ...)

► Keyword arguments allow to specify which value corresponds to which argument

► &key

► (defun my-fun (&key x y z) ...)

► (my-fun :z 14 :x 3 :y 6)

► Missing arguments are replaced by NIL

Lambda expression

- Una lambda expression è una funzione anonima che può essere usata come entità

(lambda (<argomenti>) <espressione>)

► Example

(lambda (x) (+ x 1))

→ #<FUNCTION :LAMBDA (X) (+ X 1)>

((lambda (x) (+ x 1)) 4)

→ 5

Funzioni delle liste

Il LISP considera soltanto liste singole e per lavorarci agevolmente mette a disposizione diverse funzioni:

- (list <e1> <e2>) -> crea una lista con gli argomenti inseriti

- (cons <e1> <list>) -> esegue un inserimento in testa

- (append <list1> <list2>) -> unisce le liste passate come argomento tra loro
Es. Se viene passata una lista (1, 2) ed una lista (3, 4) otterremo un'unica lista (1, 2, 3, 4)

- (null <list>) -> ritorna NULL se la lista è vuota

- (car <list>) -> ritorna la testa della coda

- (cdr <list>) -> ritorna la coda senza il suo primo elemento

► car and cdr can be composed

(caddr '(1 2 3 4))

► means

(car (cdr (cdr '(1 2 3 4))))

→ 3

Examples

(setq l (list 1 2 3 4))

→ (1 2 3 4)

(car l)

→ 1

(cdr l)

→ (2 3 4)

(append l '(5 6))

→ (1 2 3 4 5 6)

(setq l (list 1 2 3 4))

→ (1 2 3 4)

(car l)

→ 1

(cdr l)

→ (2 3 4)

(append l '(5 6))

→ (1 2 3 4 5 6)

Examples – defining functions

► Member of a list

```
(defun membro (el l)
  (if (null l)
      NIL ;; not found
      (if (equal el (car l))
          T ;; found
          (membro el (cdr l)))
          ;; recursive search
      )
  )
)
```

Le Macro

Sono costrutti simili alle funzioni, con la differenza che queste vengono usate per ripetere piccole porzioni di codice e che traducono i loro argomenti in espressioni simboliche.

Possiamo utilizzare queste macro per aggiungere nuove funzionalità nel linguaggio, grazie alle macro diventa, per esempio, possibile creare dei costrutti simile ai loop presenti in altri linguaggi.

Examples (2)

```
(defun fooF (x) (print x))
→ FOOF
(defmacro fooM (x) (print x))
→ FOOM
(fooF (+ 2 3)) → 5
(fooM (+ 2 3)) → (+ 2 3)
```

Diagram illustrating macro expansion:

- (fooF (+ 2 3)) → 5
Printed
- (fooM (+ 2 3)) → (+ 2 3)
Printed

Annotations:
→ 5 → Result of evaluation
→ (+ 2 3) → Result of evaluation

Examples (3)



► Defining a macro to manage an if construct with multiple statements

```
(defmacro if-seq (cond then else)
  `(if ,cond (progn ,@then) (progn ,@else)))
```

► You can call

```
(if-seq <condition>
<list_true_statements>
<list_false_statements>)
```

Loop

- loop → esegue ripetutamente fino a che non si arriva ad un return
- loop for → itera per x volte, non ha bisogno di return per interrompersi
- do
- dotimes → itera un numero specifico di volte
- dolist → itera su ogni elemento di una lista

Loop statements - examples

► Simple loop

```
(setq a 0)
(loop
  (setq a (+ a 1))
  (print a)
  (if (> a 9) (return a))
)
```

► For-like loop

```
(loop for x in '(pipetto pluto paperino)
      do (print x)
)
```

► Loop on a list (similar to previous)

```
(dolist (x '(pipetto pluto paperino))
  (print x)
)
```

Loop statements - examples

► The factorial with loop

```
(defun factLoop (n)
  (let ((f 1)) ; local variable
    (loop
      (setq f (* f n))
      (if (= n 1) (return f) ; true
          (setq n (- n 1)) ; else
        )
      )
  )
)
```

Altre funzioni utili

- **time** -> ritorna il tempo di esecuzione di un'espressione
 - ▶ (time <s-expression>)
 - ▶ (time (fact 10))
- **numberp** -> ritorna vero se l'argomento è un numero
 - ▶ (numberp <arg>)
 - ▶ (numberp (car 1))
- **progn** -> è in grado di raggruppare diverse dichiarazioni, valutandole poi in sequenza e ritornando il valore dell'ultima
 - ▶ (progn <st1> <st2> <st3> ...)
 - ▶ (if (not (null 1))
 - ▶ (progn (print 1) (print (car 1))))
- **every** -> prende in ingresso un'altra funzione, ritorna poi TRUE se la funzione è vera per ogni elemento della lista (every <function> <list>)
 - ▶ (every <function> <list>)
 - ▶ (every 'numberp '(1 23 7)) → T
- **subset** -> accetta gli stessi valori di every (una funzione ed una lista) e ritorna solo i valori per cui quella funzione è TRUE
 - ▶ (subset <function> <list>)
 - ▶ (subset 'numberp '(a b 3 c 7)) → (3 7)
- **mapcar** -> stessi input di every e subset, applica la funzione passata ad ogni elemento della lista
 - ▶ (mapcar <function> <list>)

L'operatore function

- L'operatore **function** accetta un nome o una lambda expression e ritornano una funzione oggetto.
- L'operatore **funcall** accetta una funzione oggetto e la applica agli argomenti
 - ▶ Example:

```
(defun adder (x) (function (lambda (y) (+ x y))))  
(setq add5 (adder 5))  
(funcall add5 2)  
→ 7
```
 - ▶ Note that add5 is a **closure** of the function on the argument 5

Java 8

Nel 2014 nasce Java 8 che apporta modifiche significative al linguaggio. Tra le più importanti l'introduzione delle lambda expression che permette di utilizzare il linguaggio seguendo uno stile di programmazione funzionale senza però trattare mai le funzioni come entità di prima classe.

Lambda expression in Java8 -> vi sono varie notazioni :

general notation -> (params) -> {body}
 | |
 argomenti Istruzioni da eseguire
 con il tipo

```
(int x) -> { return x+1; }  
(int x, int y) -> { int z= x+y;  
          return z; }  
(String s) -> {return s + "a"; }
```

shortcut notation -> (params) -> expression

- in questo caso il valore di ritorno è il valore dell'espressione dopo la sua valutazione

```
(int x) -> {System.out.println(x); }
```

omitted notation -> param -> body

- se viene passato solo un argomento le parentesi possono essere omesse

x -> x+1

Reference methods

- Java 8 introduce il simbolo ' :: ' che permette di specificare il metodo e non la sua chiamata :

```
System.out.println() = System.out::println
```

NB. Per gli esempi 1 e 2 che sono puramente esplicativi non mi sembrava necessario fare il riassunto faccio quindi riferimento alle slide per questa parte.

Iterazione interna VS Iterazione esterna

Iteratore esterno

```
public class IterationExamples {  
    public static void main(String[] args){  
        List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});  
  
        for(String letter: alphabets){  
            System.out.println(letter.toUpperCase());  
        }  
    }  
}
```

OR we can write like this:

```
public class IterationExamples {  
    public static void main(String[] args){  
        List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});  
  
        Iterator<String> iterator = alphabets.listIterator();  
        while(iterator.hasNext()){  
            System.out.println(iterator.next().toUpperCase());  
        }  
    }  
}
```

Questo tipo di iterazione ha vari problemi:

- Il for è un iteratore sequenziale che processa gli elementi nell'ordine specificato dalla stringa in questo caso
- Limita la possibilità di gestire il flusso di controllo, che potrebbe essere in grado di fornire prestazioni migliori sfruttando il riordino dei dati, il parallelismo o il cortocircuito

Iteratore interno

```
public class IterationExamples {  
    public static void main(String[] args){  
        List<String> alphabets = Arrays.asList(new String[]{"a","b","b","d"});  
  
        alphabets.forEach(l -> l.toUpperCase());  
    }  
}
```

Invece di controllare l'iterazione, il client lo lascia gestire dalla libreria e fornisce solo il codice che deve essere eseguito per tutti/alcuni dei dati.

forEach -> esegue l'azione data per ogni elemento dell'Iterable fino a quando tutti gli elementi sono stati elaborati o l'azione genera un'eccezione.
Possono anche essere utilizzati i reference methods.

```
stringhe.forEach((String s) ->{})
```

```
stringhe.forEach(System.out::println)
```

Interfacce funzionali

Java 8 introduce anche le **@FunctionalInterface** spesso rappresentano concetti astratti come funzioni, azioni o predicati e offrono inoltre bersagli per le lambda expression.

- Queste interfacce possono essere trovate in `java.util.function`, hanno un solo metodo astratto e possono avere numerosi metodi di default.

Naming convention delle interfacce funzionali

- T|U -> per il tipo dell'argomento
- R -> per il tipo del risultato
- Function -> per indicare una funzione da T a R unitaria
- Consumer -> per una funzione da T a void
- Predicate -> per una funzione da T a booleano
- Supplier -> una funzione senza parametri che fornisce R

- ▶ Functions which **extend** the basic function shapes:
`UnaryOperator` (extends `Function`),
`BinaryOperator` (extends `BiFunction`), have the same type for arguments and result
- ▶ Functions that **specialize** basic functions to primitive types: `ToIntFunction`, `DoubleConsumer`, `ObjIntConsumer`, `IntToDoubleFunction`
- ▶ `ObjIntConsumer` has one argument of type T and one argument of type int

Consumer

- ▶ We have already seen the interface `Consumer<T>`

```
package java.util.function;
import java.util.Objects;
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
    default Consumer<T> andThen(Consumer<?
super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t);
        after.accept(t); };
    }
}
```

- Il metodo `accept` deve essere implementato specificando l'azione che deve essere eseguita (t)

- Il metodo `andThen` ritorna un `Consumer` (NON una funzione)

```

import java.util.*;
import java.util.function.*;
public class LoopPrint3 {
    public static void main(String[] args) {
        ArrayList<String> strings = new ArrayList<String>();
        strings.add("one");
        strings.add("two");
        strings.add("three");

        strings.forEach(new Consumer<String>() {
            public void accept(String s) {
                System.out.println(s);
            }
        });
    }
}

```

Predicate

```

package java.util.function;
import java.util.Objects;
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    default Predicate<T> negate() {
        return (t) -> !test(t);
    }
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}

```

- L'interfaccia **Predicato** richiede che sia implementato un metodo **test** per eseguire il controllo del predicato (che avrà un booleano **true** o **false** come valore di ritorno)
- Il metodo **apply** contiene l'azione da applicare al parametro **T** per ottenere il valore di ritorno.

Function <T,R>

```

package java.util.function;
import java.util.Objects;

@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) ->
            after.apply(apply(t));
    }
    static <T> Function<T, T> identity() {
        return t -> t;
    }
}

```

- Il metodo **compose** ritorna una funzione composta che applica il **before** all'input e poi applica la funzione al risultato
- Il metodo **andThen** restituisce una funzione composta che prima applica la funzione all'input e in seguito la funzione **after**
- Il metodo **identità** restituisce sempre il suo argomento all'input.

Stream

Stream → astrazione per rappresentare l'elaborazione dei dati e permettono l'uso di architetture multicore.

Una stream è una sequenza di oggetti che supportano operazioni aggregate :

- Sono una sequenza di elementi che vengono ottenuti o elaborati su richiesta
- Provengono da collections, array, I/O
 - Le collections possono creare stream con i metodi stream() e parallelStream()
 - Gli array possono creare una stream se passati come argomento alla funzione Stream.of(), ad esempio Stream<String> s = Stream.of(a);
 - La classe stream permette di creare uno stream con i metodi generate (restituisce una stream sequenziale di elementi generati dal supplier che viene passato come argomento) e iterate (partendo da un seed come parametro, genera gli elementi usando un'UnaryOperator(T) applicandolo al seed)
- Possono venire fatte operazioni come filter, map, limit...
- Operazioni su stream restituiscono stream
- Operazioni di stream vengono eseguite su ogni elemento sorgente

Metodo forEach → che applica un azione ad ogni elemento della stream
(strings.stream().forEach()).

Metodo collect → (per terminare le operazioni di una pipeline) che mette tutti gli elementi risultanti in un container di risultati (alcuni collector di default sono contenuti nella classe Collectors).

Altri metodi sono:

Metodi Filter → restituiscono una stream con elementi che soddisfano una certa condizione (strings.stream().filter(new Predicate<String> () {public boolean test(String t) {corpo del test}}}).collect(Collectors.toList());

Metodo Map → applica un metodo a tutti gli elementi

Metodi Limit → riducono il numero degli elementi

Metodi Sorted → ordinano la stream

Metodi allMatch / anyMatch → che verificano se un predicato si applica a tutti gli elementi o solo ad alcuni