RICORSIONE:

Forma di UTILIZZO di Astrazione del CODICE per POTER Iterare in modo Alternativo

↳ Approcci diversi ma INTERSCAMBIABILI

fz RICORSIVA ⇒ Se Richiama se Stessa. (può NON Terminare)

Esempio:

```
fattoriale(int n){
  if(n<=1)
    return 1;
  else
    return n·fatt(n-1)
```

Informatica

Parametro DECRESCE e mi Garantisce la TERMINAZIONE

Assomiglia All' INDUZIONE:

$$\begin{cases} fatt(1) = fatt(0) = 1 \\ fatt(n) = n \cdot fatt(n-1) \end{cases}$$

Matematica

Così facendo NON Abbiamo dato NESSUN Vincolo per Mantenere La TOURING CALCOLABILITÀ

$fie(1) = fie(1)$ NON è INDUZIONE

Ma possiamo Scrivere un PROGRAMMA che si Comporti Così, Quindi che diverge

Altro Esempio: $\begin{cases} foo(0) = 0 \\ foo(n) = foo(n+1) \end{cases}$

Non Tende Verso la Base, Parametro Cresce ⇒ NON è INDUTTIVA

anche se ESISTE un Programma che si Comporta Così

PERCHÈ un LINGUAGGIO PERMETTA LA RICORSIONE ABBIAMO BISOGNO:

- Procedure
- Gestione Dinamica della MEMORIA (No alloc. Statica)

RICORSIONE IN CODA:

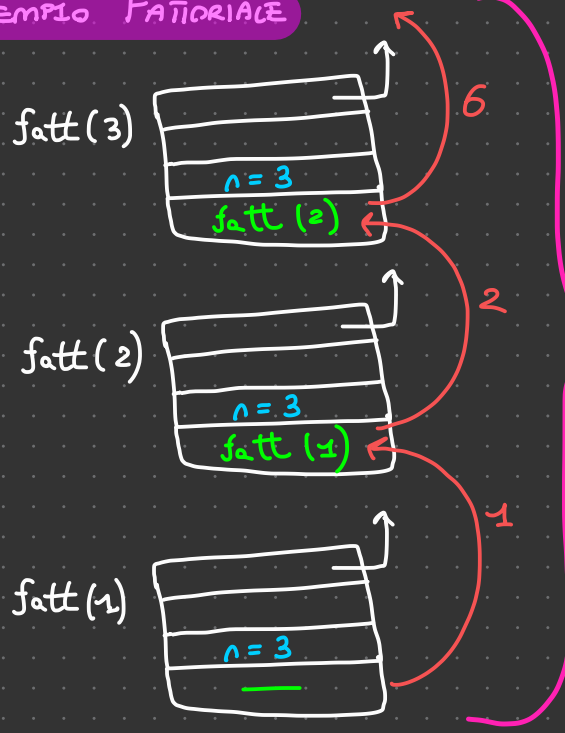
OTTIMIZZA L'USO della MEMORIA della RICORSIONE

Una CHIAMATA dentro f ad una PROCEDURA g si dice in Coda se f restituisce il Valore RITORNATO da g senza ulteriori COMPUTAZIONI

Una procedura si dice ricorsiva in CODA se la chiamata ricorsiva è in CODA.

fattoriale Viene MANIPOLATO (moltiplicato) ⇒ NON in Coda.

Esempio Fattoriale



Se NON è di coda NON Posso SOVRASCRIVERE il Record di Attivazione Precedente perchè devo MEMORIZZARE il Valore Per POI Elaborarlo.

Fattoriale In Coda:

```
int fattRC(int n, int res){
  if n<=1
     return RES
  else
     return fattRC(n-1, res·n)

int fatt(int n){
   fattRC(n,1);
```

Risultato Parziale = Caso Base All'inizio

Valore che ho Costruito Nelle Chiamate Precedenti Prima del CASO BASE

Solitamente ho Almeno 1 Parametro in Più Per la Ricorsione in Coda

n = 3

fatt(3)

→ fattRC(3, 1)

→ fattRC(2, 3)

→ fattRC(1, 6)

Sono al CASO BASE e return di RESULT che ho Costruito Nel PARAMETRO

FIBONACCI:

$Fib(0) = 1$

$Fib(1) = 1$

$Fib(n) = Fib(n-1) + Fib(n-2)$

```
int Fib(int n) {
    if n<=1
        return n;
    else
        return Fib(n-1) + Fib(n-2);
```

NON è in Coda Visto che c'è una MANIPOLAZIONE

mi Servono i 2 Passi Precedenti

```
int FibRC(int n, int res1, int res2){
    if n<1
        return res2 → Quello da RESTITUIRE/CORRETTO
    else
        return Fib(n-1, res, res1+res2)

int fib(n){
   return fibRC(n, 1, 1)
```

ESEMPIO:

fib(5) ⇝ fib(5, 1, 1)

1+1

fib(4, 1, 2)

fib(3, 2, 3)

fib(2, 3, 5)

fib(1, 5, 8)

Risultato = f(5)

TIPOLOGIE DI ESERCIZI:

- domanda di Teoria sull'Implementazione del linguaggio
  - Interpreti
  - Compilatori

  } def. Intuitiva & Formale

- Esercizi
  - Induzione Strutturale (no Matematica) (DIMOSTRARE)
  - Scoping → Catena Statica + CRT (DA COSTRUIRE)
    → Codice da Completare
  - Binding (F.Z. COME PARAMETRO)
  - Passaggio Parametri (VALORE / RIFERIMENTO)
  - Ricorsione (da NORMALE a CODA)
  - Di Semantica (ASSEGNAMENTO/DICHIARAZIONE)

COMPLETAMENTO CODICE:

Dato un **Pezzo di CODICE**:

```
{int i;
  (*)
  for(int i=0; i<1; i++){
        int x;
        x = fun();
  }
```

Fornire il Codice da inserire in (*) t.c. il linguaggio ha Scoping Statico le 2 chiamate alla funzione Restituiscono lo Stesso Valore.

Con Scoping Dinamico Invece devono Essere DIVERSE e descrivere la Scelta.

Codice deve Essere Eseguibile Quindi deve ESSERE definito tutto ciò che si Usa.

Bisogna definire una funzione fun()

Perchè il **COMPORTAMENTO Sia POTENZIALMENTE diverso** per via delle REGOLE di Scoping (AMBIENTE NON LOCALE) è Necessario che ci Sia:

- **Procedura** con **Ambiente NON Locale**
- **Riferimento** NON locale deve **∃ sia Nel Contesto di**

definizione che di Chiamata

i è un Buon Candidato per Essere un AMBIENTE non locale per fun

x NON è INIZIALIZZATA Nell' AMBIENTE DI CHIAMATA e NON può ESSERE UTILIZZATA

Se ci fossero 2 ⊛ allora avrei POTUTO inizializzarla.

Codice All' Interno di ⊛

```
i = 1; ⟶ Per Rendere CODICE Eseguibile
int fun(){
  return i;
```

↳ Ad Ogni ITERAZIONE i cambia per Ogni CHIAMATA.

ESECUZIONE SCOPING STATICO

Chiamata Restituisce la i Globale = 1 che NON Viene Modificata

ESECUZIONE SCOPING DINAMICO

Le 2 Chiamate di fun() fanno riferimento alla i del CICLO for che CAMBIA:

① return 0;

② return 1;

ALTRO ESEMPIO:

```
{ int x
  (*)
  void exec(){
      for(int J=2; J<=3; J++){
          int y = 1;
          x = fun();
      }
  }
exec();
}
```

Ambiente di Chiamata NON MODIFICABILE

È Sempre UGUALE

```
(*)
  Int J = 99;
  fun(){
    return J;
  }
```

ESECUZIONE SCOPING STATICO

J=99 viene prese Sempre IGNORANDO la Variabile J del Ciclo che Quella VARIA

ESECUZIONE SCOPING DINAMICO

Le ITERAZIONI del CICLO Portano Risultati diversi:

1. return 2;
2. Return 3;

} Se Avessi USATO y Allora Avrei Avuto Sempre RETURN 1;

ALTRO ESERCIZIO:

```
int a = 0;
(*)
{while (a <= 1){
    int x;
    (**)
    x = fun();
    a++;
  }
}
```

```
(*) int z = -10;
fun(){
  return z;
}
```

```
(**) int z = 4 + a;
```

ESECUZIONE SCOPING STATICO

Chiamata Restituisce -10 perchè si Riferisce alla z Statica

ESECUZIONE SCOPING DINAMICO

Chiamate:

1. RETURN 4;
2. RETURN 5;

a <u>NON</u> può ESSERE Usata Visto che a Esiste Solo GLOBALE e se la RIDEFINISSI potrei ROMPERE il WHILE

Ma x è una Buona Candidata (con DOVUTE Modifiche nell' Ambiente di Chiamata).

DA FARE PER CASA :

GUARDARE SLIDES