# *Ricorsione*

Luca Abeni

March 7, 2015

# Esecuzione Come Valutazione

- Programma funzionale: composto da espressioni
- “Eseguito” valutando le espressioni
- Esempio: fattoriale!

```
unsigned int fact(unsigned int n)
{
  return n == 0 ? 1: n * fact(n - 1);
}
```

  - Notare “if aritmetico” (`p ? a : b`)
  - `fact(4)` = ?

```
fact(4) = (4 == 0) ? 1 : 4 * fact(3) =
4 * fact(3) = 4 * ((3 == 0) ? 1 : 3 * fact(2)) =
4 * 3 * fact(2) = 4 * 3 * ((2 == 0) ? 1 : 2 * fact(1)) =
4 * 3 * 2 * ((1 == 0) ? 1 : 1 * fact(0)) =
4 * 3 * 2 * 1 * 1 = 24
```

# Alcune Osservazioni

- Per essere puramente funzionale, if aritmetico!

```
unsigned int fact(unsigned int n)
{
  return n == 0 ? 1: n * fact(n - 1);
}
```

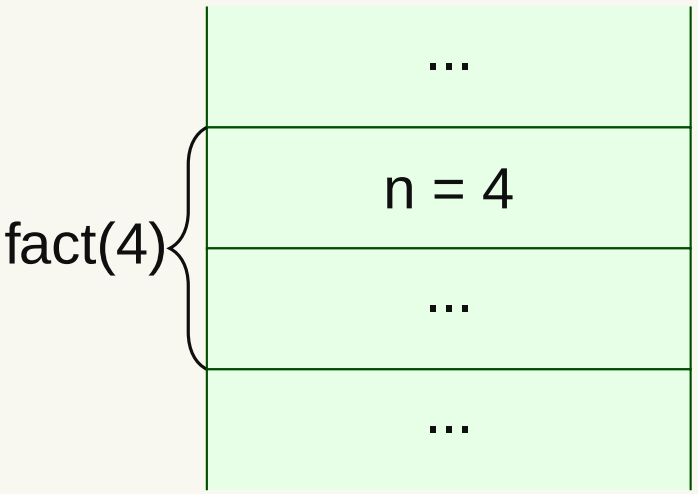
vs

```
unsigned int fact(unsigned int n)
{
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

- Esecuzione come valutazione/sostituzione possibile solo in assenza di effetti collaterali

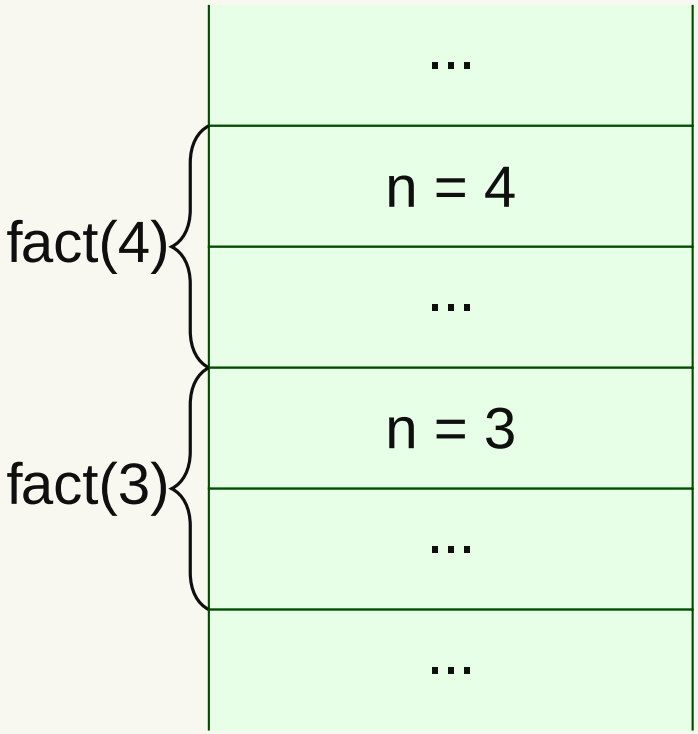
# E... Lo Stack?

- Invocazione di funzione $\rightarrow$ alloca record di attivazione sullo stack...
- Vediamo allora cosa succede allo stack durante la ricorsione!
- `fact(4)`: nuovo record di attivazione (stack frame) contenente:
  - Parametro formale $n = 4$
  - Link (statico e dinamico) a precedenti stack frame
  - Spazio per valore di ritorno



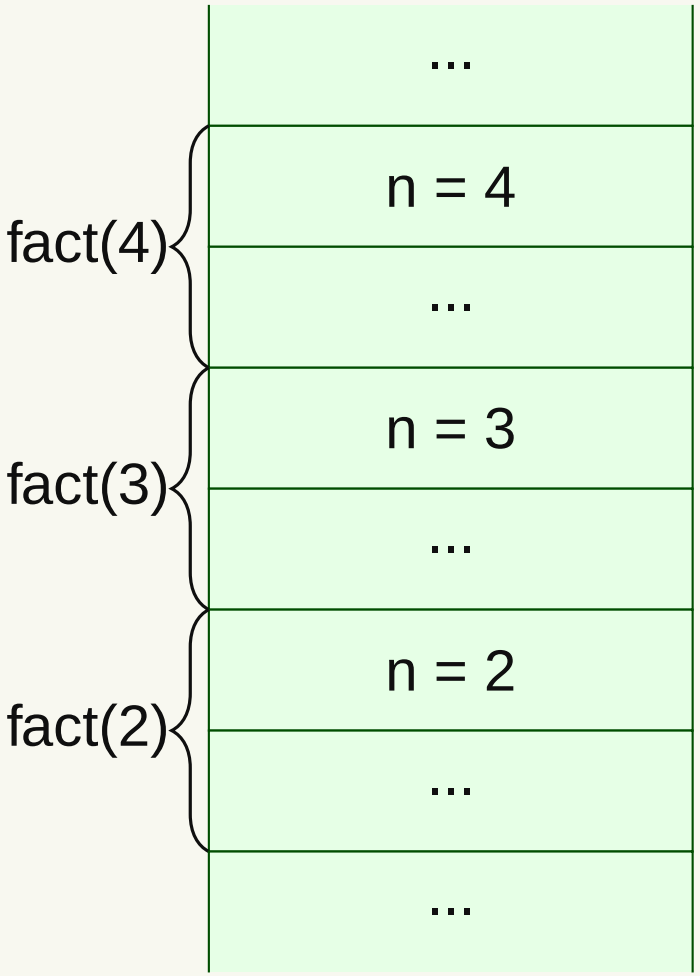
# Stack Frame - 2

- `4 * fact(3)`



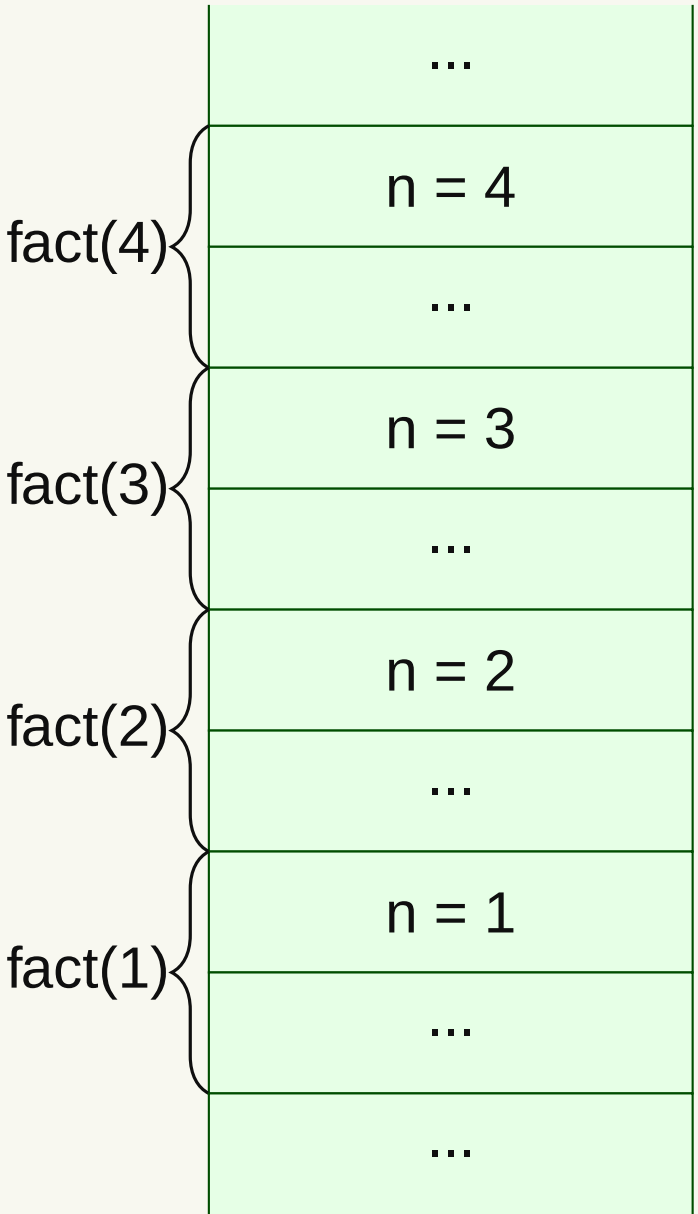
# Stack Frame - 3

- `4 * 3 * fact(2)`



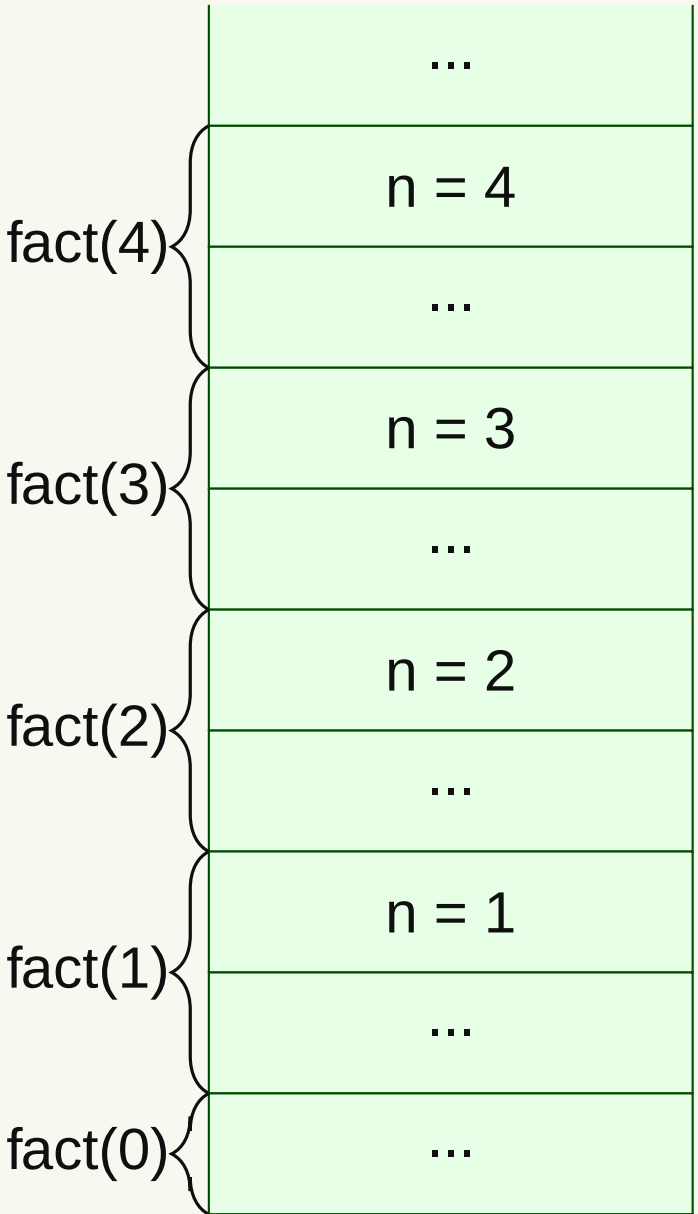
# Stack Frame - 4

- `4 * 3 * 2 * fact(1)`



# Stack Frame - 5

- `4 * 3 * 2 * 1 * fact(0)`



# Considerazioni

- Quando si arriva a valutare `fact(0)`, gli stack frame precedenti contengono i numeri da moltiplicare...
- Gli stack frame vengono rimossi uno dopo l'altro (quando le varie istanze di `fact()` terminano) eseguendo le moltiplicazioni
- Quando `fact(n - 1)` termina, `fact(n)` deve eseguire la moltiplicazione per n
  - Non può terminare immediatamente!
- I vari stack frame sono quindi necessari fino a che la relativa istanza di `fact()` non termina, e non possono essere rimossi prima dallo stack
  - Ricorsione $\Rightarrow$ grosso consumo di stack
  - Possibili stack overflow (già visti in azione!)

# Ricorsione e Stack

- Il consumo di stack è un prezzo che va sempre pagato per fare ricorsione?
- Consideriamo implementazione alternativa del fattoriale:

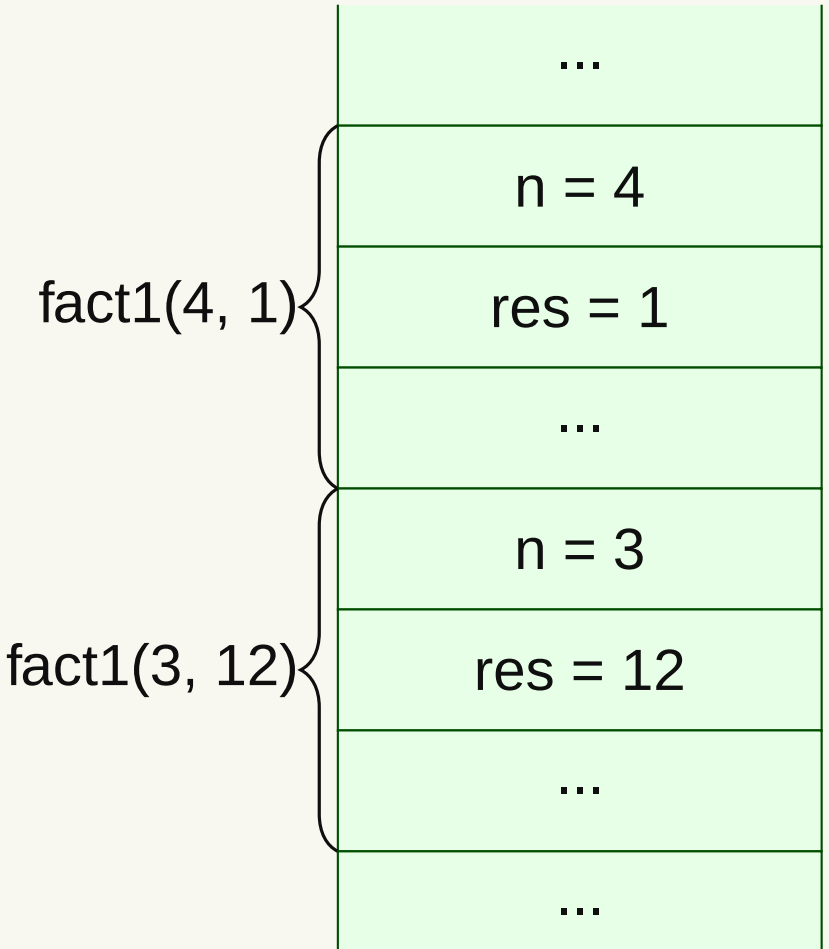
```
unsigned int fact1(unsigned int n, unsigned int res)
{
  return n == 0 ? res : fact1(n – 1, n ∗ res);
}
unsigned int fact(unsigned int n)
{
  return fact1(n, 1);
}
```

- Perché il secondo parametro???

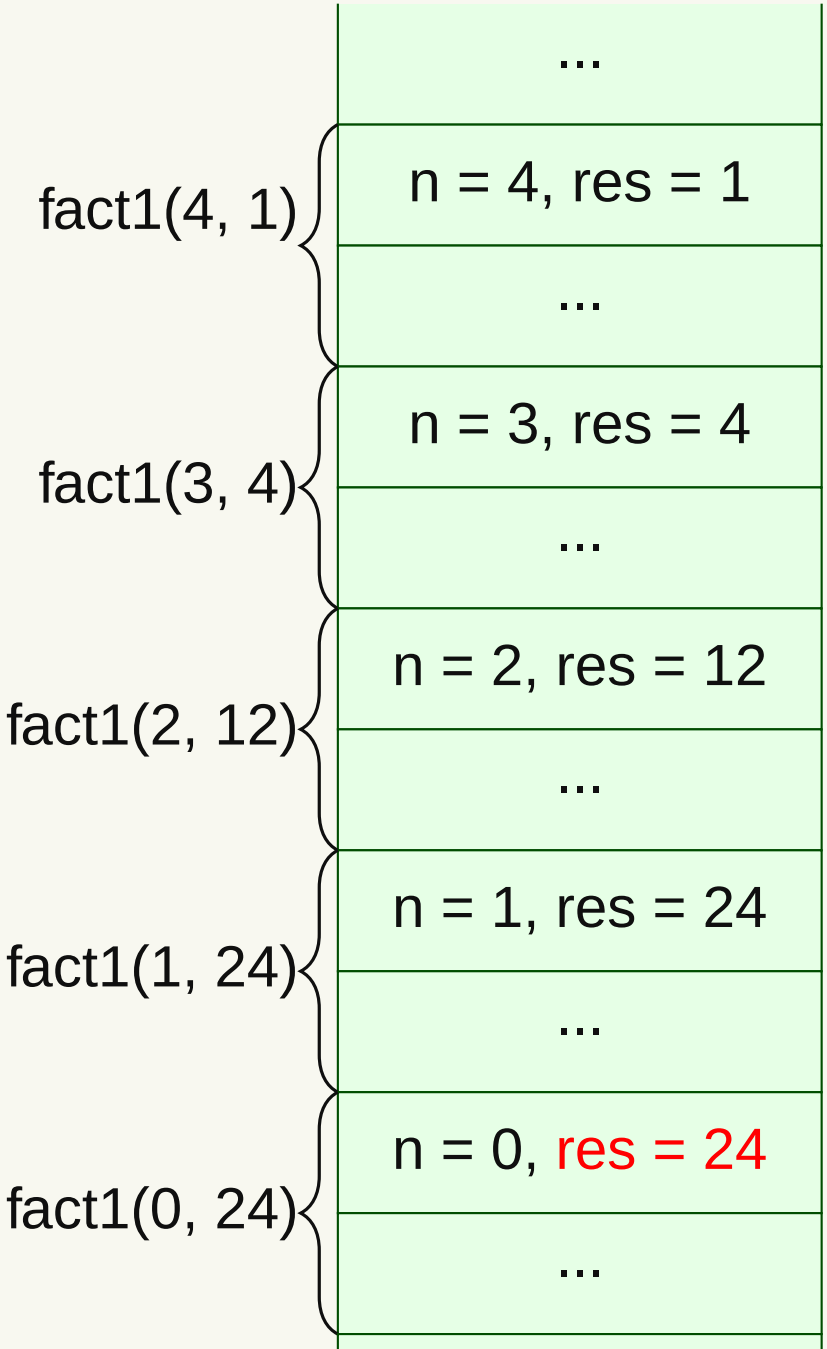
```
fact(4) = fact1(4, 1) = (4 == 0) ?  1 :  fact1(3, 4 * 1) =
fact1(3, 4) = (3 == 0) ?  4 :  fact1(2, 3 * 4) =
fact1(2, 12) = (2 == 0) ?  12 :  fact1(1, 2 * 12) =
fact1(1, 24) = fact1(0, 1 * 24) = 24
```

# Stack Frame

- Nessuna moltiplicazione da effettuare quando `fact1(n-1, ...)` ritorna...
- In ogni istante, i valori degli stack frame precedenti non contengono dati necessari / utili!



# Stack Frame - 2



# Considerazioni

- Quando si arriva a valutare `fact1(0, ...)`, gli stack frame precedenti non contengono valori utili...
- Gli stack frame vengono rimossi uno dopo l'altro (quando le varie istanze di `fact1()` terminano) senza dover eseguire ulteriori operazioni
- Quando `fact1(n - 1, ...)` termina, `fact1(n, ...)` ritorna direttamente il suo valore di ritorno
  - Può terminare immediatamente, passando il valore di ritorno al chiamante!
- I vari stack frame possono quindi essere rimossi dallo stack al momento della ricorsione (prima che la funzione associata termini)
  - Ricorsione $\Rightarrow$ nessun consumo di stack
  - No stack overflow!