# Breve Introduzione alla Programmazione Funzionale per Programmatori Imperativi

Luca Abeni

24 novembre 2021

## 1 Stili e Paradigmi di Programmazione

La programmazione funzionale è uno stile (o un *paradigma*) di programmazione; sebbene esistano linguaggi che facilitano la scrittura di programmi secondo il paradigma funzionale (o addirittura impongono l'utilizzo di tale paradigma di programmazione), è possibile scrivere programmi secondo lo stile funzionale indipendentemente dal linguaggio di programmazione che si sta utilizzando (e quindi anche usando un linguaggio tradizionalmente considerato "imperativo" come il C).

Per capire meglio cosa si intenda per programmazione funzionale (e come utilizzare uno "stile di programmazione funzionale"), consideriamo il modo in cui siamo abituati a sviluppare un programma. Siamo tradizionalmente abituati a considerare un algoritmo come una sequenza di azioni che vanno ad "operare" su qualcosa: per esempio, una ricetta di cucina può essere vista come una sequenza di azioni sugli ingredienti, che li trasformano nel piatto che vogliamo cucinare... O, più informaticamente parlando, un programma può essere visto come una sequenza di istruzioni che agiscono su uno stato condiviso (memoria/variabili, dispositivi di I/O, ...).

Vedere un programma come una sequenza di istruzioni che agiscono sulla memoria (o sui dispositivi di I/O) è consistente con l'architettura dei moderni computer (una o più CPU che eseguono istruzioni Assembly, le quali operano su una memoria condivisa) che deriva dall'architettura di Von Neumann. Esistono però anche altri modi di pensare ad un programma (o ad un algoritmo).

Consideriamo per esempio l'algoritmo di Euclide per il calcolo del massimo comun divisore (mcm, gcd in inglese). Questo algoritmo, che conosciamo fin dalle scuole medie, dice circa: "*dati due numeri naturali $a$ e $b$, se $b = 0$ allora $a$ è il massimo comun divisore. Altrimenti, si assegna ad $a$ il valore di $b$ ed $a$ $b$ il resto della divisione fra $a$ e $b$, poi si ricomincia dall'inizio.*"

Una semplice implementazione di questo algoritmo usando un approccio imperativo si ottiene "traducendolo" in un linguaggio di programmazione imperativo. Per esempio, usando la sintassi del linguaggio C si ottiene il codice mostrato in Figura 1: L'unica piccola complicazione rispetto alla descrizione dell'algoritmo è l'introduzione della variabile temporanea `tmp`, che serve a non perdere il valore di $b$ quando si assegna `b = a % b`. Il discorso diventa più complicato quando ci chiediamo *perché l'algoritmo di Euclide funziona* (vale a dire: possiamo provare che l'algoritmo implementato qui sopra calcola correttamente il massimo comun divisore fra due numeri?). La spiegazione del funzionamento dell'algoritmo è circa la seguente:

- Il massimo comun divisore fra $a$ e 0 è chiaramente $a$ (poiché 0 è divisibile per qualsiasi numero, con resto 0)
- Il massimo comun divisore fra $a$ e $b \neq 0$ è uguale al massimo comun divisore fra $b$ ed $a\%b$ (dimostrabile per induzione)

Dal punto di vista matematico, questo significa che

$$gcd(a, b) = \begin{cases} \text{a} & \text{se b} = 0 \\ \text{gcd(b, a \% b)} & \text{altrimenti} \end{cases}$$

e questa definizione matematica ci porta ad una nuova implementazione dell'algoritmo di Euclide mostrata in Figura 2. I puristi obietteranno che questa implementazione non è strutturata (ha un unico punto di ingresso ma 2 diversi punti di uscita: ci sono 2 diversi statement `return`) e questo "problema" può essere risolto usando il cosiddetto "if aritmetico", come mostrato in Figura 3.

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  while (b != 0) {
    unsigned int tmp;

    tmp = b;
    b = a % b;
    a = tmp;
  }

  return a;
}
```

Figura 1: Algoritmo di Euclide implementato in C.

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  if (b == 0) {
    return a;
  }

  return gcd(b, a % b);
}
```

Figura 2: Algoritmo di Euclide implementato in modo ricorsivo.

```c
unsigned int gcd(unsigned int a, unsigned int b)
{
  return (b == 0) ? a : gcd(b, a % b);
}
```

Figura 3: Implementazione puramente funzionale dell'algoritmo di Euclide.

Confrontando l'implementazione di Figura 1 (che verrà chiamata "implementazione imperativa") con quella di Figura 3 (che verrà chiamata "implementazione funzionale"), si possono notare due differenze importanti:

- Mentre l'implementazione imperativa dell'algoritmo lavora modificando il valore di alcune variabili (locali) "`a`", "`b`" e "`tmp`", l'implementazione funzionale non modifica il valore di alcuna variabile (in altre parole, non utilizza l'operatore "`=`" di assegnamento)
- Mentre l'implementazione imperativa utilizza un ciclo "`while(b != 0)`", l'implementazione funzionale utilizza il meccanismo della ricorsione (con la condizione "`b == 0`" come condizione di fine ricorsione / base induttiva)

Si noti come la seconda caratteristica dell'implementazione funzionale sia una conseguenza più o meno diretta della prima: un ciclo "`while`" è basato su un predicato che viene valuatato (ripetendo l'esecuzione del corpo del ciclo) fino a che non diventa falso. Il valore di questo predicato (nell'esempio, "`b != 0`") è calcolato in base al valore di una o più variabili ("`b`" nell'esempio precedente); quindi, se il valore di tali variabili non può essere modificato il valore di verità del predicato sarà sempre vero o sempre falso, rendendo praticamente inutilizzabile il costrutto di ciclo (che può dare origine solo a cicli infiniti o mai eseguiti). Per questo motivo, in assenza di variabili modificabili i costrutti di iterazione (`while` e simili) non possono essere utilizzati.

In generale, un'implementazione funzionale di un algoritmo sarà basata su *funzioni pure*, vale a dire funzioni intese nel senso matematico del termine (relazioni $f \subset \mathcal{D} \times \mathcal{C}$ fra un insieme dominio $\mathcal{D}$ ed un insieme del codominio $\mathcal{C}$ che ad ogni elemento del dominio assegnano alpiù un elemento del codominio), senza alcun tipo di *effetto collaterale*. Formalmente, $f(x) = y$ significa $(x, y) \in f$ ed indica che:

```c
int f(int v)
{
  static int acc;

  acc = acc + 1;

  return v + acc;
}
```

Figura 4: Esempio di funzione non pura, con effetti collaterali.

- la funzione $f$ associa sempre lo stesso valore $y \in \mathcal{C}$ al valore $x \in \mathcal{D}$
- calcolare $y$ a partire da $x$ è l'unico effetto della funzione

Poiché la modifica del valore di una variabile è un effetto collaterale dell'operatore di assegnamento, questo significa che **il paradigma di programmazione funzionale non prevede il concetto di variabili modificabili**. Come conseguenza, in un programma scritto secondo lo stile funzionale non esistono cicli, che sono sostituiti da chiamate ricorsive. Inoltre, non si usano comandi con effetti collaterali, ma solo espressioni che ritornano un valore e l'esecuzione di un programma non avviene modificando uno stato ma valutando espressioni. Questo è il motivo per cui usando il paradigma funzionale non si utilizza il costrutto condizionale `if` (che seleziona l'esecuzione alternativa di due diversi comandi) ma il cosiddetto *if aritmetico* (il costrutto "`... ? ... : ...`" in C), che genera un risultato valutando una fra due diverse espressioni dipendentemente dal valore di un predicato. La prima conseguenza di questo fatto è che in un programma funzionale i rami "`else`" dei costrutti "`if`" devono essere sempre presenti.

Successive valutazioni della stessa espressione (o invocazioni della stessa funzione pura con gli stessi parametri) devono risultare nello stesso risultato. Sebbene questo requisito possa sembrare banale e scontato (è insito nella definizione matematica di funzione), non è verificato ogni qual volta ci si trovi in presenza di effetti collaterali. Per esempio, si consideri la funzione C di Figura 4: successive invocazioni `f(2); f(2); f(2);` ritorneranno valori diversi (2, 3 e 4). Per capire come mai questo possa essere un problema, si consideri l'espressione `(f(2) + 1) * (f(2) + 5)`: se l'invocazione a sinistra è valutata per prima, il risultato è $(2+1)*(3+5) = 24$, altrimenti è $(3+1)*(2+5) = 28$.

# 2 Ricorsione ed Iterazione

Mentre i costrutti base della programmazione imperativa sono (secondo l'approccio strutturato) la sequenza di comandi, il costrutto di selezione (esecuzione condizionale, `if`) ed il ciclo (per esempio, `while`), i costrutti base della programmazione funzionale sono l'invocazione di funzione, l'operatore di if aritmetico e la ricorsione.

In particolare, l'equivalente funzionale dell'iterazione (ciclo) è la ricorsione: ogni algoritmo che codificato secondo il paradigma imperativo richiede un ciclo (finito o infinito), codificato secondo il paradigma funzionale risulta in una ricorsione (ancora, finita o infinita).

La tecnica della ricorsione (strettamente legata al concetto matematico di *induzione*) è usata in informatica per definire un qualche genere di "entità"[1] basata su se stessa. Focalizzandosi sulle funzioni ricorsive, si può definire una funzione $f()$ esprimendo il valore di $f(n)$ come funzione di altri valori calcolati da $f()$ (tipicamente, $f(n-1)$).

In generale, le definizioni ricorsive sono date "per casi", vale a dire sono composte da varie clausole. Una di queste è la cosiddetta *base* (detta anche *base induttiva*); esistono poi una o più clausole o *passi induttivi* che permettono di generare / calcolare nuovi elementi a partire da elementi esistenti. La base è una clausola della definizione ricorsiva che non fa riferimento all'"entità" che si sta definendo (per esempio: il massimo comun divisore fra $a$ e 0 è $a$, etc...) ed ha il compito di porre fine alla ricorsione: senza una base induttiva, si da' origine ad una ricorsione infinita.

In sostanza, una funzione $f : \mathcal{N} \to \mathcal{X}$ è definibile definendo una funzione $g : \mathcal{N} \times \mathcal{X} \to \mathcal{X}$, un valore $f(0) = a$ ed imponendo che $f(n+1) = g(n, f(n))$. Più nei dettagli, una funzione è definibile per ricorsione quando ha come dominio l'insieme dei naturali (o un insieme comunque numerabile); il codominio può essere invece un generico insieme $\mathcal{X}$. Come base induttiva, si definisce il valore della funzione per il più

[1] Il termine "entità" è qui usato informalmente per indicare genericamente funzioni, insiemi, valori, tipi di dato, ...

```
unsigned int fattoriale(unsigned int n)
{
  if (n == 0) {
    return 1;
  }

  return n * fattoriale(n - 1);
}
```

Figura 5: Implementazione ricorsiva della funzione fattoriale.

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

Figura 6: Implementazione funzionale della funzione fattoriale.

piccolo valore facente parte del dominio (per esempio, $f(0) = a$, con $a \in \mathcal{X}$) e come passo induttivo si definisce il valore di $f(n+1)$ in base al valore di $f(n)$; come detto sopra, questo si può fare definendo $f(n+1) = g(n, f(n))$). Notare che il dominio di $g()$ è l'insieme delle coppie di elementi presi dal dominio e dal codominio di $f()$, mentre il codominio di $g()$ è uguale al codominio di $f()$.

L'esempio tipico portato sempre quando si parla di ricorsione è la funzione fattoriale, che può essere codificata in modo ricorsivo come mostrato in Figura 5. Una versione più propriamente funzionale (perché utilizza solo espressioni, e non comandi) del fattoriale è mostrata invece in Figura 6

Questo esempio ci può essere utile per vedere come l'esecuzione di un programma scritto secondo il paradigma funzionale possa essere vista come una sequenza di "semplificazioni" o "sostituzioni" (tecnicamente, *riduzioni*) analoghe a quelle fatte per valutare un'espressione aritmetica. Si consideri per esempio il calcolo di `fattoriale(4)`. Dopo che la funzione `fattoriale()` è stata definita (come sopra, per esempio), nell'ambiente esiste un legame fra il nome "fattoriale" ed un'entità denotabile (il corpo della funzione fattoriale). Di fronte all'espressione "`fattoriale(4)`" è quindi possibile cercare nell'ambiente il legame al corpo della funzione e sostituire "`fattoriale`" con la sua definizione, usando "`4`" (parametro attuale) al posto del parametro formale "`n`":

fattoriale (4) $\rightarrow$ (4 == 0) ? 1 : 4 * fattoriale (4 $-$ 1) $\rightarrow$ 4 * fattoriale (3)

dove il primo passaggio corrisponde sostanzialmente alla sostituzione del nome "`fattoriale`" col corpo della funzione (secondo il legame trovato nell'ambiente) e la sostituzione del parametro formale "`n`" con il valore "`4`" in tale corpo, mentre il secondo passaggio è avvenuto perché $4 \neq 0$ (quindi, si valuta la seconda sottoespressione "4 * fattoriale (4 $-$ 1)") e $4-3=1$. A questo punto, si cerca ancora il nome "`fattoriale`" nell'ambiente e si applica il corpo della funzione al parametro attuale "`3`":

4 * fattoriale (3) $\rightarrow$ 4 * ((3 == 0) ? 1 : 3 * fattoriale (3 $-$ 1)) $\rightarrow$ 4 * (3 * fattoriale (2))

procedendo analogamente si ottiene

4 * (3 * fattoriale (2)) $\rightarrow$ 4 * (3 * ((2 == 0) ? 1 : 2 * fattoriale (2 $-$ 1))) $\rightarrow$

$\rightarrow$ 4 * (3 * (2 * fattoriale (1))) $\rightarrow$ 4 * (3 * (2 * ((1 == 0) ? 1 : 1 * fattoriale (1 $-$ 1)))) $\rightarrow$

$\rightarrow$ 4 * (3 * (2 * (1 * fattoriale (0)))) $\rightarrow$

$\rightarrow$ 4 * (3 * (2 * (1 * ((0 == 0) ? 1 : 0 * fattoriale (0 $-$ 1))))) $\rightarrow$

$\rightarrow$ 4 * (3 * (2 * (1 * 1))) = 24.

Si noti come dal punto di vista logico il calcolo del fattoriale abbia richiesto solo operazioni di:

1. ricerca nell'ambiente (per poter applicare una funzione ai suoi argomenti / parametri attuali bisogna trovare nell'ambiente il binding fra il nome della funzione ed il suo corpo)

2. sostituzione testuale (l'applicazione di una funzione ai suoi argomenti si ottiene per semplice sostituzione dei parametri attuali al posto dei parametri formali nel corpo della funzione trovato al punto 1)

3. calcolo aritmetico (questo include sia l'esecuzione di operazioni aritmetiche "semplici" come prodotti e sottrazioni che la valutazione di if aritmetici)

Questo mostra come secondo il paradigma funzionale l'esecuzione di un programma avvenga per riduzione, vale a dire per sostituzione testuale di espressioni e sotto-espressioni: una funzione applicata ad un argomento è sostituita dal corpo della funzione ed il parametro formale è sostituito dal parametro attuale. Concettualmente, questo processo di riduzione non richiede l'esecuzione di istruzioni Assembly o di programmi, ma solo manipolazioni di stringhe come quelle operabili (per esempio) con un editor di testo (in più, sarà necessario effettuare i calcoli richiesti dalle varie operazioni aritmetiche - quindi, volendo continuare con l'analogia di cui sopra si può dire come siano in realtà necessari un editor di testo ed una calcolatrice).

Chiaramente, questo concetto di computazione come riduzione è applicabile solo a funzioni pure, vale a dire senza effetti collaterali: se `fattoriale()` avesse effetti collaterali (come per esempio la modifica di variabili globali, o simili) non sarebbe possibile sostituire semplicemente "fattoriale (4)" con "4 $*$ fattoriale (3)". Questo spiega perché l'assenza di effetti collaterali è (come già anticipato) un requisito fondamentale per il paradigma di programmazione funzionale; l'eliminazione di variabili modificabili è un modo semplice per eliminare tutta una grande classe di effetti collaterali (rimangono gli effetti collaterali dovuti a I/O, ma questi spesso non possono essere eliminati senza rendere inutile il programma).

Il paradigma di programmazione funzionale, che è stato presentato in queste pagine come alternativa al "tradizionale" paradigma imperativo, ha lo stesso potere espressivo del paradigma imperativo (vale a dire: qualsiasi algoritmo codificabile usando un approaccio imperativo può essere implementato anche usando l'approccio funzionale). I lettori più abituati a sviluppare programmi secondo l'approccio imperativo potranno obiettare che "rinunciare" alle variabili modificabili ed all'iterazione sembra complicare lo sviluppo dei programmi e che di conseguenza l'approccio funzionale può apparire un po' innaturale. In realtà questo è più un problema di abitudine ed una volta che si capisce la logica della programmazione funzionale lo sviluppo di programmi secondo questo approccio risulterà più semplice. Inoltre, esistono problemi che sono più facilmente risolubili usando la ricorsione e che non sono propriamente semplici da risolvere usando un approccio puramente imperativo. Per esempio, consideriamo il problema delle torri di Hanoi.

Il problema consiste nello spostare una torre composta da $N$ dischi (di dimensioni decrescenti) da un palo ad un altro, usando un terzo palo "di appoggio" per gli spostamenti. Le regole del gioco prevedono che si possa spostare un solo disco per volta e che non si possa appoggiare un disco più grande sopra ad uno più piccolo. Mentre sviluppare una soluzione non ricorsiva al problema non è semplicissimo (e richiede di utilizzare strutture dati complesse), una soluzione ricorsiva è banale. In pratica, il problema di spostare $N$ dischi dal palo $a$ al palo $b$ (usando il palo $c$ come appoggio) è scomponibile nel problema di:

- Spostare $N-1$ dischi dal palo $a$ al palo $c$
- Spostare il rimanente disco (il più grande) dal palo $a$ al palo $b$
- Spostare gli $N-1$ dischi dal palo $c$ al palo $b$

Ora, mentre il secondo passo dell'algoritmo (spostare un disco da un palo all'altro) è semplice, il primo ed il terzo passo richiedono di spostare $N-1$ dischi e non sono direttamente implementabili. Ma se sappiamo come spostare $N$ dischi, possiamo utilizzare (ricorsivamente!) lo stesso algoritmo per spostare $N-1$ dischi (e questo richiedera di spostare $N-2$ dischi, poi un disco e poi ancora $N-2$ dischi). E questa ricorsione può essere invocata più volte (per la precisone, $N-1$ volte) fino a che il problema non è ridotto allo spostamento di un solo disco.

La Figura 7 mostra una semplice implementazione di questo algoritmo usando il linguaggio C. Si noti come in questo caso la condizione di fine ricorsione (base induttiva) corrisponda allo spostamento di un singolo disco (come nell'algoritmo descritto poc'anzi). Un'implementazione alternativa avrebbe potuto usare la condizione "`n == 0`" (nessun disco da spostare) come base induttiva, risultando nell'implementazione della funzione `move()` mostrata in Figura 8.

Un'importante considerazione da fare sul codice proposto è che sebbene utilizzi la ricorsione non è ancora implementato secondo un approccio puramente funzionale: la funzione `move()` utilizza infatti una sequenza di comandi (`move()` e `move_disk()` non hanno alcun valore di ritorno) basando il proprio funzionamento sugli effetti collaterali di tali comandi (la stampa a schermo tramite `printf()`). Per implementare `move()` come una funzione pura, bisognerebbe eliminarne gli effetti collaterali (vale a dire, eliminare la chiamata a `printf()` da `move_disk()`, aggiungendo un valore di ritorno a `move()` e `move_disk()`). La soluzione più ovvia è quella di fare si che `move()` e `move_disk()` ritornino una stringa

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void move_disk(const char *from, const char *to)
{
  printf("Muovi disco da %s a %s\n", from, to);
}

void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 1) {
    move_disk(from, to);

    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  unsigned int height = 0;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  move(height, "Left", "Right", "Center");

  return 0;
}
```

Figura 7: Soluzione del problema delle torri di Hanoi.

```c
void move(unsigned int n, const char *from, const char *to, const char *via)
{
  if (n == 0) {
    return;
  }

  move(n - 1, from, via, to);
  move_disk(from, to);
  move(n - 1, via, to, from);
}
```

Figura 8: Soluzione alternativa del problema delle torri di Hanoi.

contenente il loro output (in altre parole, la sequenza di mosse da fare per risolvere il problema non deve essere stampata a schermo tramite `printf()`, ma salvata in una stringa).

```
const char *concat(const char *a, const char *b)
{
  char *res;

  res = malloc(strlen(a) + strlen(b) + 1);
  memcpy(res, a, strlen(a));
  memcpy(res + strlen(a), b, strlen(b));
  res[strlen(a) + strlen(b)] = 0;

  return res;
}

const char *move_disk(const char *from, const char *to)
{
  return concat(concat(concat(concat("Move disk from ", from),
                " to "), to), "\n");
}

const char *move(int n, const char *from, const char *to, const char *via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      concat(concat(move(n - 1, from, via, to),
                move_disk(from, to)), move(n - 1, via, to, from));
}
```

Figura 9: Soluzione funzionale del problema delle torri di Hanoi.

Una possibile soluzione (purtroppo non leggibilissima a causa della sintassi del linguaggio C) è mostrata in Figura 9. In questa soluzione, la funzione di utilità `concat()` riceve in ingresso due stringhe (in C, array di caratteri) e ritorna una stringa che contiene (come il nome suggerisce) la concatenazione delle due. Ci sono un po' di cose importanti da notare:

- L'utilizzo della notazione funzionale per `concat()` (invece di un operatore infisso, come in altri linguaggi) riduce la leggibilità del codice. Il lettore non deve però essere confuso dalle lunghe catene "`concat(concat(concat(...)))`", che non fanno altro che concatenare lunghe sequenze di stringhe
- Le funzioni `move()` e `move_disk()` sono ora *funzioni pure*, in quanto non fanno affidamento su effetti collaterali
- Gli effetti collaterali sono ora tutti concentrati nella funzione `main()`, che effettua I/O... E' chiaro che se il programma deve comunicare coll'ambiente esterno da qualche parte dovrà effettuare I/O, che è un effetto collaterale; un programma non potrà quindi mai essere "puramente funzionale", ma tenderà a concentrare tutti gli effetti collaterali in punti specifici (per i linguaggi funzionali, il ciclo Read-Evaluate-Print o il supporto runtime)
- A conferma del fatto che sono funzioni pure, `move()` e `move_disk()` non modificano il contenuto di variabili (non usano assegnamenti, o altri comandi, ma sono composte solo da espressioni)
- I lettori più attenti si saranno accorti del fatto che il programma presenta dei memory leak: `concat()` alloca dinamicamente memoria per la stringa che ritorna, ma tale memoria non viene mai liberata. Questo fatto è una consegenza del punto precedente (il contenuto delle stringhe non viene mai modificato, ma la concatenazione di due stringhe avviene allocando dinamicamente la memoria per la stringa risultato) e mostra come il paradigma di programmazione funzionale richieda un *garbage collector* (che viene infatti sempre incluso nelle macchine astratte che implementano linguaggi di programmazione funzionali)

```
#include <cstdlib>
#include <iostream>
#include <string>

std::string move_disk(std::string from, std::string to)
{
  return "Move disk from " + from + " to " + to + "\n";
}

std::string move(int n, std::string from, std::string to, std::string via)
{
  return (n == 1) ?
      move_disk(from, to)
    :
      move(n - 1, from, via, to) +
                move_disk(from, to) + move(n - 1, via, to, from);
}

int main(int argc, char *argv[])
{
  int height = 0;
  std::string res;

  if (argc > 1) {
    height = atoi(argv[1]);
  }
  if (height <= 0) {
    height = 8;
  }

  res = move(height, "Left", "Right", "Center");

  std::cout << res;

  return 0;
}
```

Figura 10: Soluzione funzionale in C++ del problema delle torri di Hanoi.

Per mostrare che molte delle complicazioni che appaiono nel programma precedente non sono dovute allo stile di programmazione funzionale, ma alla sintassi del linguaggio C (ed alla sostanziale mancanza di funzioni per la gestione delle stringhe) la Figura 10 mostra una reimplementazione in C++ (che fornisce un supporto per le stringhe molto più avanzato rispetto al C), che appare molto più pulita.

# 3 Ricorsione e Stack

Come precedentemente visto, per valutare un'espressione tramite sostituzione/riduzione non è concettualmente necessario introdurre i concetti di invocazione di subroutine, stack, record di attivazione e simili. D'altra parte, se la macchina astratta che esegue il nostro programma (o, valuta la nostra espressione) è implementata su un'architettura hardware basata sul modello di Von Neumann (come tutti i moderni PC) dovrà per forza utilizzare il meccanismo di chiamata a subroutine (istruzione Assembly `call` su architetture Intel, etc...) per invocare l'esecuzione di una funzione[2].

[2] Anche implementazioni alternative, che interpretano la funzione invece di compilarla in Assembly per rimanere più fedeli al modello di valutazione per sostituzione visto in precedenza, dovranno utilizzare strutture dati a pila (o simile) che crescono ad ogni applicazione della funzione.

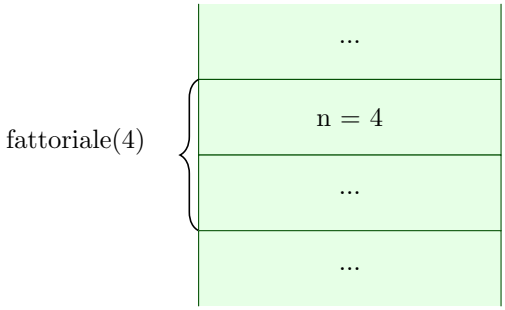


Figura 11: Stack frame per l'invocazione `fattoriale(4)`.

Tornando all'esempio del fattoriale visto in precedenza, ogni volta che la funzione `fattoriale()` invoca se stessa (ma questo vale anche per la generica invocazione di altre funzioni) dovrà quindi essere aggiunto un nuovo record di attivazione (o stack frame) sullo stack, facendone crescere la dimensione. In particolare, ad ogni invocazione ricorsiva della funzione verrà aggiunto sullo stack un record di attivazione contenente il parametro attuale con cui `fattoriale()` è stata invocata, un link dinamico (ma in alcuni linguaggi anche un link statico) al precedente stack frame ed un po' di spazio per memorizzare il valore di ritorno. Quando si invoca "`fattoriale(4)`" si crea quindi la situazione visibile in Figura 11.

Poiché anche "`fattoriale(3)`" si invocherà ricorsivamente, lo stack evolve come mostrato in Figura 12, crescendo ad ogni invocazione ricorsiva. Questo comporta che il calcolo del fattoriale di un numero `n` abbastanza grande richiederà una grande quantità di memoria. Si noti che il record di attivazione corrispondente a "`fattoriale(n)`" non può essere rimosso dallo stack fino a che "`fattoriale(n - 1)`" non è terminata, perché contiene il valore "`n`" per cui il risultato di "`fattoriale(n - 1)`" deve essere moltiplicato. In sostanza, quando si arriva a valutare "`fattoriale(0)`, gli stack frame precedenti contengono i numeri da moltiplicare; man mano che le varie istanze di `fattoriale()` terminano, gli stack frame vengono rimossi dallo stack uno dopo l'altro (dopo aver eseguito la moltiplicazione per il valore di "`n`" contenuto nello stack frame). I vari stack frame sono quindi necessari fino a che la relativa istanza di "`fattoriale`" non termina, e non possono essere rimossi prima dallo stack.

Questo problema sembrerebbe compromettere la reale utilizzabilità delle tecniche di programmazione funzionale, in quanto lunghe catene di chiamate ricorsive porterebbero ad un consumo di memoria eccessivo (mentre lunghe iterazioni hanno generalmente un consumo di memoria basso e costante). Per capire meglio questa cosa, si consideri il problema di implementare una funzione che testa se un numero naturale è pari o dispari senza usare operazioni di divisione o modulo. Una possibile soluzione è mostrata in Figura 13. La soluzione proposta usa una mutua ricorsione fra le funzioni `pari()` e `dispari()`, basandosi sull'idea che 0 è pari, 1 è dispari (basi induttive) ed ogni numero $n > 1$ è pari se $n - 1$ è dispari o è dispari se $n - 1$ è pari (passo induttivo). A parte la bizzarria di questa soluzione, si può immediatamente immaginare come l'invocazione di `pari()` o `dispari()` su numeri grandi possa finire per causare una grossa crescita dello stack (fino allo stack overflow). Infatti, se si compila il programma con `gcc paridispari.c` e se ne testa il funzionamento per numeri piccoli tutto sembra funzionare... Ma provando con numeri abbastanza grandi (si provi per esempio 24635743, come suggerito nel commento) si ottiene un segmentation fault dovuto a stack overflow.

La cosa sorprendente è però che provando a compilare il programma con `gcc -O2 paridispari.c` il programma funziona correttamente con qualsiasi numero si immetta in ingresso! Questo accade perché il problema della crescita dello stack è facilmente aggirabile usando la cosiddetta "tail call optimization" (ottimizzazione delle chiamate in coda, abilitata dallo switch "`-O2`" di `gcc`), che permette, sotto opportune ipotesi, di sostituire invocazioni di funzioni con semplici salti (trasformando quindi una ricorsione in un'iterazione).

Per capire meglio come funziona questa ottimizzazione, consideriamo la versione "tail recursive" del fattoriale, mostrata in Figura 14. Intuitivamente, si può vedere come la funzione `fattoriale_tr()` utilizzi un secondo argomento per "accumulare" il risultato: la moltiplicazione per "`n`" avviene *prima* della chiamata ricorsiva (per calcolare il valore del secondo parametro attuale) e non dopo. Questo comporta che il valore "`n`" non deve essere salvato per essere utilizzato quando la chiamata ricorsiva termina... L'espressione "`fattoriale(4)`" viene valutata come segue:

$\text{fattoriale}\ (4) \rightarrow\ \text{fattoriale\_tr}\ (4,\ 1) \rightarrow (4 == 0)\ ?\ 1 :\ \text{fattoriale\_tr}\ (4 - 1,\ 4 * 1) \rightarrow$
$\rightarrow\ \text{fattoriale\_tr}\ (3,\ 4) \rightarrow (3 == 0)\ ?\ 4 :\ \text{fattoriale\_tr}\ (3 - 1,\ 3 * 4) \rightarrow$
$\rightarrow\ \text{fattoriale\_tr}\ (2,\ 12) \rightarrow (2 == 0)\ ?\ 12 :\ \text{fattoriale\_tr}\ (2 - 1,\ 2 * 12) \rightarrow$

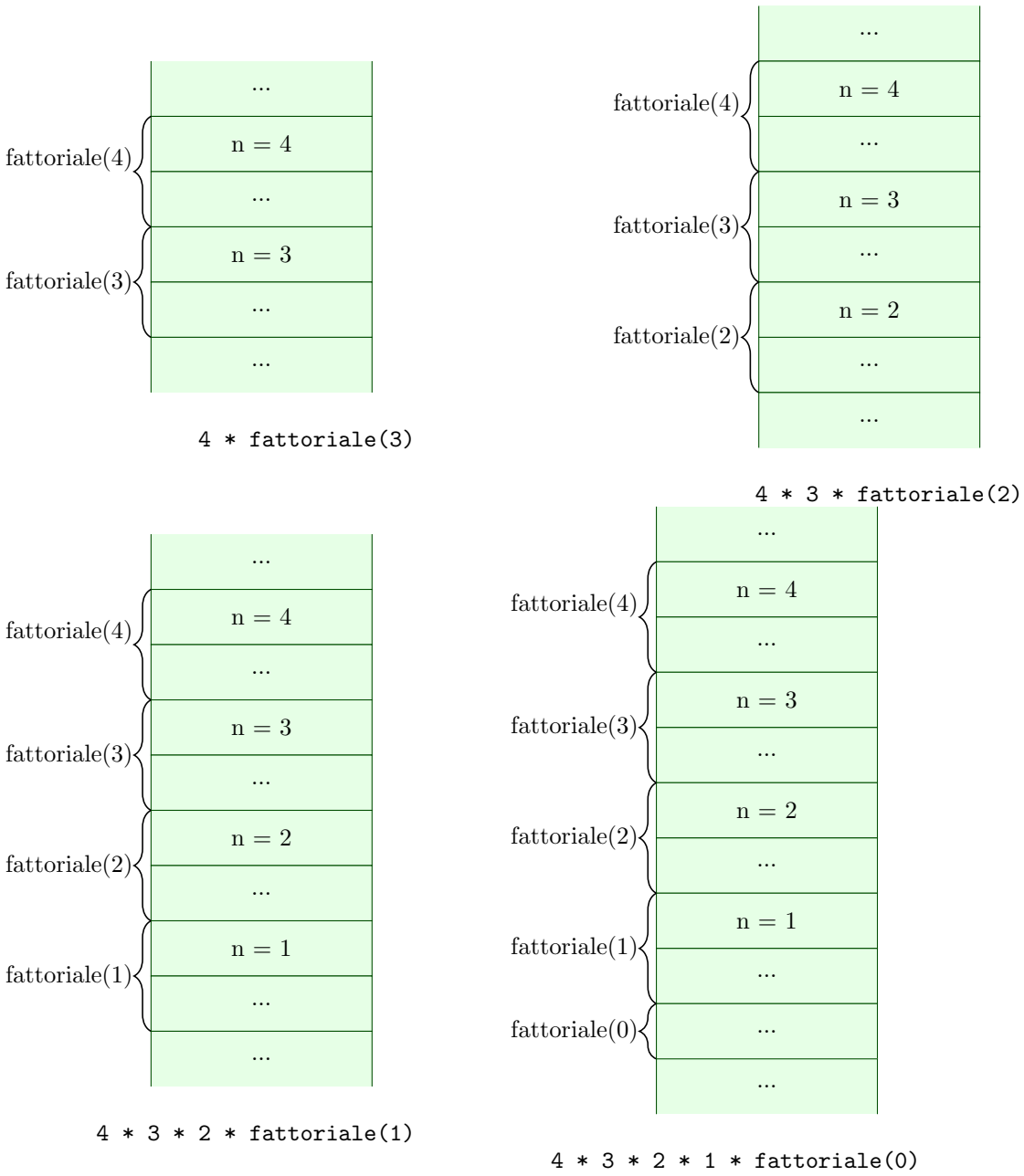


Figura 12: Evoluzione dello stack per l'invocazione `fattoriale(4)`.

```c
/* Try 24635743... */

unsigned int pari(unsigned int n);
unsigned int dispari(unsigned int n)
{
  if (n == 0) return 0;
  return pari(n - 1);
}

unsigned int pari(unsigned int n)
{
  if (n == 0) return 1;
  return dispari(n - 1);
}
```

Figura 13: Funzioni (mutuamente) ricorsive per testare se un numero è pari o dispari.

```c
unsigned int fattoriale_tr(unsigned int n, unsigned int res)
{
  return (n == 0) ? res : fattoriale_tr(n - 1, n * res);
}

unsigned int fattoriale(unsigned int n)
{
  return fattoriale_tr(n, 1);
}
```

Figura 14: Versione tail recursive del fattoriale.

→ fattoriale_tr (1, 24) → (1 == 0) ? 24 : fattoriale_tr (1 − 1, 1 ∗ 24) →
→ fattoriale_tr (0, 24) → (0 == 0) ? 24 : fattoriale_tr (0 − 1, 0 ∗ 24) → 24

L'osservazione fondamentale qui è che quando "`fattoriale_tr(0, 24)`" ritorna, "`fattoriale_tr(1, 24)`" può immediatamente ritornare il valore da essa ritornato... E così pure "`fattoriale_tr(2, 12)`", "`fattoriale_tr(3, 4)`" e "`fattoriale_tr(4, 1)`". Quindi, non è più necessario salvare sullo stack i record di attivazione per contenere il valore di "`n`", il valore di ritorno e l'indirizzo di ritorno: "`fattoriale_tr(0, 24)`" può direttamente ritornare il risultato 24 al chiamante originario. La Figura 15 mostra i dettagli dell'evoluzione dello stack derivante dall'invocazione di "`fattoriale(4)`", chiarendo ancora di più che durante l'esecuzione di un'istanza di `fattoriale_tr()` i record di attivazione corrispondenti alle precedenti istanze contengono dati che rimangono inutilizzati (risultando quindi essere non necessari / inutili!!!). In sostanza, quando si arriva a valutare "`fattoriale_tr(0, 24)`", tutti i dati necessari al calcolo sono contenuti nei parametri attuali ed i record di attivazione di "`fattoriale_tr(1, 24`"..."`fattoriale_tr(4, 1)`" contenuti sullo stack non vengono acceduti. Tali record di attivazione vengono rimossi dallo stack uno dopo l'altro (quando le varie istanze di `fattoriale_tr()` terminano) senza dover eseguire ulteriori operazioni su di essi: quando "`fattoriale_tr(n - 1, ...)`" termina, "`fattoriale_tr(n, ...)`" ritorna direttamente il suo valore di ritorno, senza eseguire ulteriori operazioni. Questo significa che quando la chiamata ricorsiva ritorna, ogni istanza di `fattoriale_tr()` può terminare immediatamente, passando direttamente al proprio chiamante il valore di ritorno ricevuto dall'invocazione ricorsiva. I vari stack frame possono quindi essere rimossi dallo stack al momento della ricorsione (prima che la funzione associata termini), trasformando di fatto una chiamata ricorsiva in un semplice salto.

Questa ottimizzazione è possibile ogni volta che una funzione ritorna come valore di ritorno il risultato ottenuto dall'invocazione di un'altra funzione (in sostanza, "`return altrafunzione(...)`"): in pratica, statement del tipo "**return** f(n);" non vengono compilati come invocazioni alla subroutine `f()`, ma come salti al corpo di tale funzione. Questo permette di implementare la ricorsione senza causare eccessivi consumi di stack, rendendo praticabile l'utilizzo del paradigma di programmazione funzionale (a patto di scrivere codice che utilizzi chiamate in coda).

Per capire come funziona la tail call optimization in pratica, si consideri il codice Assembly x86_64

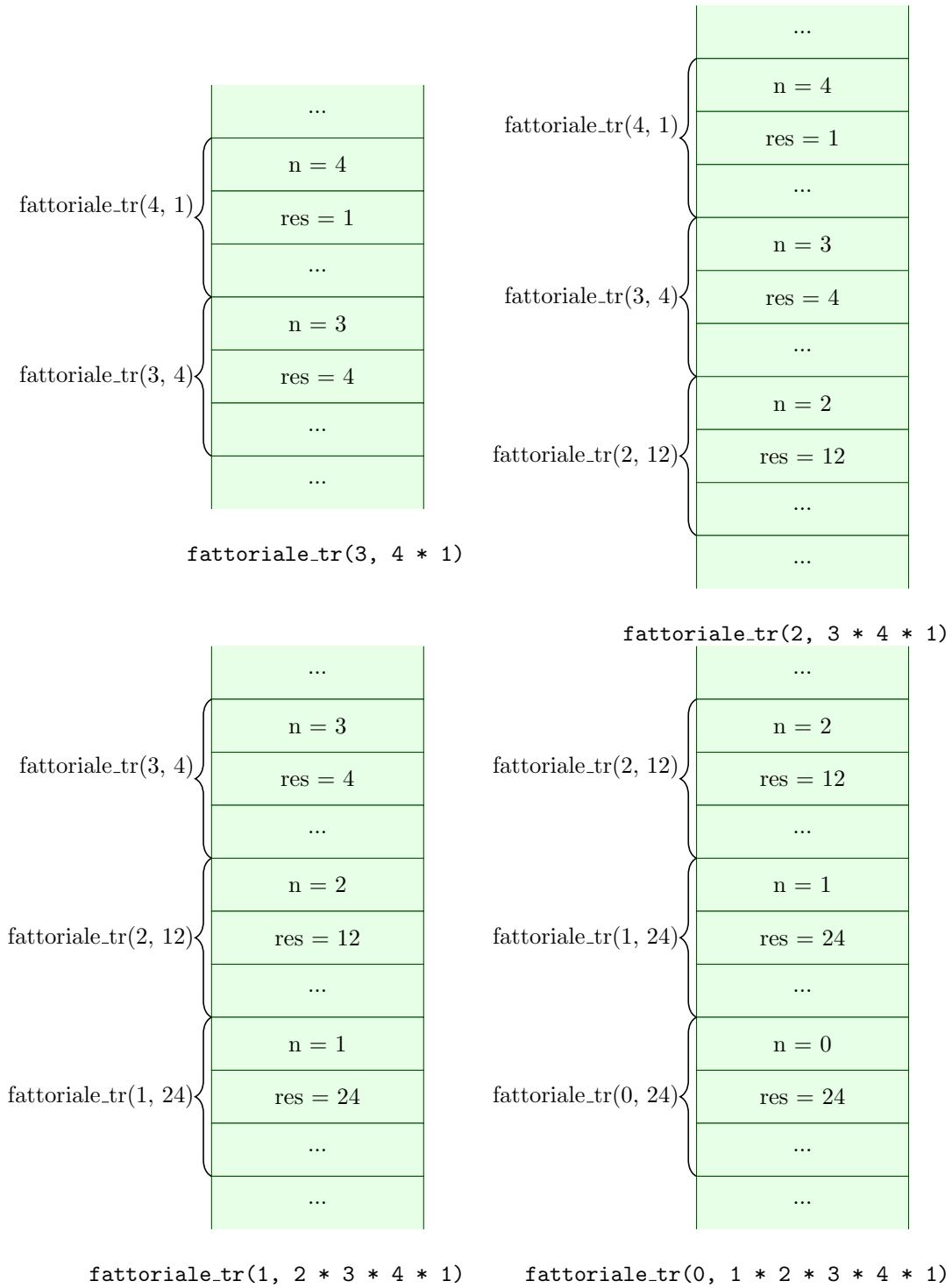


Figura 15: Evoluzione dello stack per l'invocazione `fattoriale_tr(4)`.

```
fattoriale_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        subq    $8, %rsp
        imull   %edi, %esi
        subl    $1, %edi
        call    fattoriale_tr
        addq    $8, %rsp
.L2:
        ret
```

Figura 16: Versione tail recursive del fattoriale compilata senza ottimizzazioni.

```
fattoriale_tr:
        movl    %esi, %eax
        testl   %edi, %edi
        je      .L2
        imull   %edi, %esi
        subl    $1, %edi
        jmp     fattoriale_tr
.L2:
        ret
```

Figura 17: Versione tail recursive del fattoriale ottimizzata.

generato da `gcc -O1` quando compila la funzione `f1()` di Figura 14. Questo codice è mostrato in Figura 16.

La prima istruzione (`movl`) copia il secondo argomento (contenuto nel registro `%esi`) nel registro `%eax` (che verrà utilizzato per il valore di ritorno); la seconda istruzione (`testl`) controlla se il primo argomento (contenuto nel registro `%edi`) è 0: in questo caso, si salta subito all'uscita della funzione (label `.L2`) ritornando il valore contenuto in `%eax` (nel quale è appena stato copiato il secondo argomento). Se invece il primo argomento è $> 0$, la funzione moltiplica il secondo argomento per il primo e poi si invoca ricorsivamente (le istruzioni `subq` ed `addq` applicate ad `%rsp` sono necessarie a causa delle convenzioni di chiamata dell'ABI intel a 64 bit). Si noti che al ritorno dalla chiamata ricorsiva (istruzione successiva alla `call`) non vengono eseguite altre istruzioni e la funzione termina immediatamente. E' allora possibile evitare di pushare sullo stack successivi indirizzi di ritorno inutili (quando una istanza di `fattoriale_tr()` ritorna, tutte le precedenti istanze ritorneranno immediatamente, "a catena", senza frapporre istruzioni Assembly fra le varie "`ret`"). Questo può essere fatto semplicemente eliminando le istruzioni che manipolano lo stack (registro `%rsp`) e sostituendo la `call fattoriale_tr` con una `jmp fattoriale_tr`, come mostrato in Figura 17.

# 4 Funzioni e Spezie

In matematica, siamo abituati a considerare funzioni $f : \mathcal{D} \to \mathcal{C}$ che mappano uno o più elementi del dominio $\mathcal{D}$ in alpiù un elemento del codominio $\mathcal{C}$. In pratica, $f$ è una relazione (sottoinsieme $f \subset \mathcal{D} \times \mathcal{C}$ delle coppie aventi il primo elemento in $\mathcal{D}$ ed il secondo elemento in $\mathcal{C}$) per cui $(x_1, y_1) \in f \wedge (x_1, y_2) \in f \Rightarrow y_1 = y_2$. Se $f$ ha più di un argomento, il dominio $\mathcal{D}$ è rappresentato come prodotto cartesiano di altri insiemi: per esempio, una funzione da coppie di numeri reali in numeri reali sarà $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R}^3$.

Dal punto di vista informatico, siamo invece abituati a considerare funzioni a più argomenti in cui un argomento è rappresentato da un diverso parametro formale. Per esempio, "**int** f(**int** a, **float** x, **unsigned int** z)" è una funzione che riceve un argomento di tipo intero, un argomento che rappresenta l'approssimazione di un numero reale ed un terzo argomento intero positivo (vale a dire, un numero naturale); si potrebbe quindi dire che è equivalente ad una funzione $f : \mathcal{Z} \times \mathcal{R} \times \mathcal{N} \to \mathcal{Z}$. In alcuni linguaggi di programmazio-

[3] tipicamente, nei corsi avanzati di analisi si considerano funzioni $f : \mathcal{R}^n \to \mathcal{R}^m$).

```c
int sommainteri(int a, int b)
{
  return a + b;
}
```

Figura 18: Funzione C che somma 2 interi.

```c
int (* somma_c(int a))(int b)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figura 19: Tentativo di currying della funzione `sommainteri()` (Figura 18) usand il linguaggio C.

ne è possibile utilizzare un tipo *tupla* per raggruppare tutti gli argomenti e rappresentare meglio valori appartenenti a $\mathcal{Z} \times \mathcal{R} \times \mathcal{N}$.

Questo non è però l'unico modo di rappresentare funzioni a più argomenti: in particolare, è stato mostrato da diversi matematici come qualsiasi funzione a più argomenti sia rappresentabile usando funzioni ad un solo argomento. Per esempio, usando la cosiddetta tecnica del *currying*[4] una funzione ad $n$ argomenti è rappresentabile come una "catena" di funzioni aventi tutte un solo argomento. Il "trucco" per ottenere questo risultato è che ogni funzione di questa "catena" ha come valore di ritorno una funzione (e non un "valore semplice"). Per esempio, una funzione $f : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ è rappresentabile come $f : \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$.

Questo fatto ha alcune importanti conseguenze, sia teoriche che pratiche. La prima conseguenza teorica è che un formalismo matematico (come per esempio il $\lambda$-calcolo) che considera solo funzioni aventi un unico argomento può essere perfettamente generico, a patto che queste funzioni possano generare valori di tipo funzione come risultato ed accettare valori di tipo funzione come argomenti (tali funzioni sono spesso chiamate *funzioni di ordine superiore*). La seconda conseguenza, che ha ricadute anche pratiche, è quindi l'introduzione di *funzioni di ordine superiore*, vale a dire funzioni che possono manipolare valori di tipo funzione. Per questo motivo nei linguaggi di programmazione funzionale c'è un'uniformità fra codice e dati, nel senso che le funzioni possono essere valori memorizzabili ed esprimibili[5].

E' importante notare come dal punto di vista informatico la tecnica del currying sia utilizzabile in presenza di funzioni che generano funzioni (e non semplici puntatori a funzione) come valore di ritorno. Questo fatto ne preclude, per esempio, l'utilizzo in linguaggi come il C. Per capire questa cosa, si consideri la funzione `sommainteri()` mostrata in Figura 18 e si provi a generarne una "forma curryificata". Tale funzione dovrebbe ricevere in ingresso un intero `a` e generare come risultato una funzione che dato un intero `b` somma `a` a `b`. Utilizzando una piccola estensione al linguaggio C implementata da gcc, che permette di annidare le definizioni di funzioni, si potrebbe pensare di codificare la cosa come in Figura 19. A parte la scarsa leggibilità che ci è ancora una volta regalata dalla sintassi del linguaggio C (ed il fatto che si è utilizzata una definizione di `s()` annidata dentro alla definizione di `somma_c()`, cosa non permessa dal linguaggio C standard), questo codice implementa una funzione `somma_c` che riceve un parametro formale `a` di tipo `int` e ritorna un puntatore ad una funzione che riceve un parametro formale di tipo `int` e ritorna un valore di tipo `int`. La funzione è quindi utilizzabile come mostrato in Figura 20, dove `f` è una variabile di tipo puntatore a funzione da intero a intero.

Sebbene questo codice sia compilabile con gcc (e sembri addirittura funzionare correttamente in alcuni casi), contiene un grosso errore concettuale: `somma_c()` ritorna un *puntatore* alla funzione `s()` che utilizza una variabile non locale (ad `s()`) `a`. Tale variabile è il parametro attuale di `somma_c`, che sta memorizzato sullo stack durante il tempo di vita di `somma_c`... Ma viene rimosso quando `somma_c` termina! L'invocazione `f(2)` andrà quindi ad accedere a memoria non inizializzata, generando risultati non

[4] Si noti che il nome di questa tecnica deriva da Haskell Curry, non da una spezia!

[5] Si ricorda che un valore è memorizzabile quando può essere assegnato ad una variabile ed è esprimibile quando può essere generato come risultato di un'espressione.

```
int main()
{
  int (*f)(int b);

  f = somma_c(3);

  printf("3 + 2 = %d\n", f(2));

  return 0;
}
```

Figura 20: Utilizzo della funzione `somma_c()` di Figura 19.

```
int main()
{
  int (*f1)(int b);
  int (*f2)(int b);

  f1 = somma_c(3);
  f2 = somma_c(4);

  printf("3 + 2 = %d\n", f1(2));
  printf("4 + 2 = %d\n", f2(2));

  return 0
}
```

Figura 21: Problema con la funzione `somma_c()` di Figura 19.

definiti (undefined behaviour). Sebbene qualche semplice test possa funzionare correttamente, l'esempio di Figura 21 mostrerà tutti i limiti della soluzione precedentemente proposta.

Il problema può essere risolto solo facendo si che la nostra funzione `somma_c()` ritorni una *vera funzione* (comprensiva anche del suo ambiente non locale!) e non semplicemente un puntatore a funzione. Tecnicamente, questa cosa è implementabile usando una *chiusura*, vale a dire una coppia "(ambiente, puntatore a funzione)" dove l'ambiente dovrà contenere il binding fra il nome "`a`" ed una variabile non allocata sullo stack. La soluzione tipica è di allocare nello heap un record di attivazione che contiene le variabili non locali riferite nella chiusura; questo chiaramente crea dei rischi di memory leak (che con la "tradizionale" allocazione dei record di attivazione sullo stack non si incontrano) e rende necessaria l'implementazione di un garbage collector (quando la chiusura non è più utilizzata, il record di attivazione allocato sullo heap può essere deallocato). D'altra parte, abbiamo già visto in precedenza come il paradigma di programmazione funzionale renda necessaria la presenza di un garbage collector.

Utilizzando una sintassi del tipo "`<tipo1> -> <tipo2>`" per rappresentare una funzione che riceve un argomento di tipo "`<tipo1>`" e ritorna un risultato di tipo "`<tipo2>`", la soluzione corretta al problema di cui sopra (codificare la "forma curryificata" della funzione `sommainteri()` di Figura 18) è mostrata in Figura 22. In generale, una funzione "*tipo3* `f(`*tipo1* `a, ` *tipo2* `b)`" soggetta a currying diventa "*tipo2*`->`*tipo3* `fc(`*tipo1*`a)`" tale che `f(a,b) = (fc(a))(b)` (dal punto di vista matematico, $f(a,b) \rightarrow f'(a) = f_a : f_a(b) = f(a,b)$). Se una funzione ha più di 2 argomenti, si rimuovono uno alla volta usando il currying.

Riassumendo, funzioni a più argomenti e funzioni di ordine superiore ad un solo argomento hanno lo stesso potere espressivo; molti linguaggi di programmazione funzionale, fornendo funzioni di ordine superiore si limitano ad un solo argomento / parametro formale. ML ed Haskell adottano questo approccio: anche se esiste una sintassi semplificata che permette di definire funzioni come se avessero più argomenti (per esempio, usando la keyword `fun` in Standard ML) questa viene poi convertita dalla macchina astratta nella definizione della funzione "curryificata". Per esempio, in Standard ML **fun** f p1 p2 p3 ... = exp; è equivalente a **val rec** f = **fn** p1 => **fn** p2 => **fn** p3 . . . => exp;.

Un esempio che può essere utile per capire meglio il concetto di currying è quello della funzione derivata: si consideri l'implementazione di una funzione `calcoladerivata()` che calcola la derivata

```
int->int somma_c(int a)
{
  int s(int b) {
    return a + b;
  }

  return s;
}
```

Figura 22: Currying corretto della funzione `sommainteri()` (Figura 18) usando uno pseudo-lingugaggio simile al C.

```
double calcoladerivata(double (*f)(double x), double x)
{
  const double delta = 0.001;

  return (f(x) - f(x - delta)) / delta;
}
```

Figura 23: Calcolo della derivata di una funzione in un punto, in C.

```
double->double derivata(double f(double x))
{
  double f1(double x)
  {
    const double delta = 0.001;

    return (f(x) - f(x - delta)) / delta;
  }

  return f1;
}
```

Figura 24: Calcolo della derivata di una funzione.

(meglio: il rapporto incrementale sinistro per un piccolo valore di $\delta$) di una data funzione in un punto specificato. La funzione riceve quindi come argomenti la funzione $f : \mathcal{R} \to \mathcal{R}$ di cui calcolare la derivata ed il punto $x \in \mathcal{R}$ in cui calcolare la derivata, ritornando il valore $d \in \mathcal{R}$ della derivata. Una semplice implementazione di `calcoladerivata()` usando il linguaggio C può per esempio essere quella mostrata in Figura 23.

Si provi ora ad implementare una funzione `derivata()`, simile alla precedente, ma che ritorna la funzione derivata (ancora: il rapporto incrementale sinistro per un piccolo valore di $\delta$) invece che calcolarne il valore in un punto. La funzione `derivata()` riceve quindi come argomento una funzione $f : \mathcal{R} \to \mathcal{R}$ e ritorna una funzione $f' : \mathcal{R} \to \mathcal{R}$; poiché il valore di ritorno deve essere una funzione (completa del proprio ambiente non locale) e non un semplice puntatore a funzione, `derivata()` non può essere implementata in C (vedere al proposito il precedente esempio con `sommainteri()`). Usando un linguaggio simile al C ma che supporta funzioni di ordine superiore (con la sintassi descritta in precedenza), si può implementare come mostrato in Figura 24. I lettori più attenti si saranno sicuramente accorti del fatto che `derivata()` non è nient'altro che la forma "curryificata" di `calcoladerivata()`[6]!

A titolo di esempio, in Figura 25 si riporta l'implementazione di `derivata()` in C++ (usando le estensioni funzionali fornite da C++11, come le lambda expression). Dal codice, si possono vedere alcune cose interessanti. Prima di tutto, la classe "`std::function`" fornita dal linguaggio C++ (a partire dallo standard C++11) permette di utilizzare una sintassi più semplice ed intuitiva di quella dei puntatori a funzione del C. Inoltre, tale classe non memorizza semplicemente un puntatore a funzione (o una

[6]Provando a re-implementare le due funzioni con un qualsiasi linguaggio funzionale, la cosa diventa ancora più evidente

```
#include <iostream>
#include <functional>

double f(double x)
{
  return x * x + 2 * x + 1;
}

std::function<double (double)> derivata(std::function<double (double)> f)
{
  const double epsilon = 0.0001;

  return [epsilon, f](double x) {
    return (f(x + epsilon) - f(x)) / epsilon;
  };
}

int main()
{
  double x = 2;
  std::function<double (double)> f1;

  std::cout << "f'(" << x << ") = " << (derivata(f))(x) << std::endl;

  f1 = derivata(f);
  std::cout << "f'(" << x << ") = " << f1(x) << std::endl;

  return 0;
}
```

Figura 25: Calcolo della derivata di una funzione in C++.

funzione), ma un'intera chiusura (formata dalla funzione e dal suo ambiente); tale chiusura memorizzerà (all'interno dell'oggetto di classe "`std::function`") i valori di `epsilon` ed `f`. Per finire, è da notare come la sintassi "`[epsilon,f](double x)`" permetta di definire una *funzione anonima*, che viene memorizzata (assieme al suo ambiente) nel valore di ritorno di "`derivata()`" (questo costrutto è chiamato "lamba function", per motivi che diventeranno chiari leggendo le prossime sezioni).

# 5 Linguaggi di Programmazione Funzionale

Riassumendo quanto detto fin qui, il paradigma di programmazione funzionale si contraddistingue per l'assenza del concetto di variabili modificabili (in modo da eliminare gli effetti collaterali ad essi legati), il conseguente utilizzo della ricorsione al posto dell'iterazione ed il fatto che i programmi siano composti da espressioni e non da comandi (che hanno effetti collaterali). Un'importante conseguenza della mancanza di effetti collaterali è la possibilità di eseguire i programmi usando un meccanismo di riduzione / sostituzione invece che modificando lo stato della macchina astratta. Le espressioni e funzioni diventano inoltre entità esprimibili e memorizzabili, portanto ad un'altra interessante caratteristica della programmazione funzionale: la presenza di funzioni di ordine superiore (funzioni che possono operare su altre funzioni, ricevendo funzioni come argomenti e generando funzioni come risultati). L'utilizzo di funzioni di ordine superiore diventa addirittura necessario se si limita ad uno il numero di possibili argomenti per una funzione (utilizzando la tecnica del currying per implementare funzioni a più argomenti).

Sebbene questo stile di programmazione sia utilizzabile anche con linguaggi "più tradizionali", esistono dei linguaggi, detti *linguaggi di programmazione funzionali* che cercano di favorirne (o addirittura forzarne) l'utilizzo. Caratteristica fondante di questa classe di linguaggi è quindi il tentativo di ridurre al minimo gli effetti collaterali: sebbene alcuni linguaggi funzionali prevedano il concetto di variabile modificabile, essi possono essere utilizzati anche senza fare uso di tale costrutto; alcuni linguaggi fun-

zionali (come per esempio Haskell), poi, non prevedono proprio l'esistenza di variabili modificabili. Tali linguaggi sono detti *linguaggi funzionali puri*. Altra caratteristica fondamentale dei linguaggi funzionali è poi la possibilità di trattare in modo omogeneo codice e dati (oltre ai tradizionali tipi di dato esiste il "tipo funzione", esistono funzioni di ordine superiore, etc...).

In questa sede non vengono discusse le specifiche sintassi e/o semantiche dei vari linguaggi di programmazione funzionale, ma per tali dettagli si rimanda a specifici documenti.

Nei linguaggi funzionali esiste quindi un ambiente che contiene legami (binding) fra nomi e valori (di tipi scalari, strutturati, o funzione). Tali legami vengono creati quando si invoca una funzione (legame fra parametro formale ed espressione passata come parametro attuale), ma poiché è spesso utile (anche se non strettamente necessario - ma di questo parleremo poi) avere anche un ambiente non locale ad alcuna funzione ogni linguaggio di programmazione funzionale fornisce qualche modo per associare nomi a valori in un ambiente globale. Un linguaggio funzionale fornisce quindi:

- Un sistema di tipi (type system), vale a dire un insieme di tipi predefiniti (e di valori per questi tipi), più un insieme di operatori per combinare valori dei vari tipi costruendo espressioni ed una serie di regole per assegnare un tipo ad ogni valore e verificare la corretta tipizzazione di ogni espressione;
- Un qualche modo per definire funzioni, vale a dire un meccanismo di *astrazione* che data un'espressione la astrae rispetto al valore di un parametro formale (tipicamente si considera un solo argomento e si usa il currying). Questo meccanismo è spesso fornito da un operatore che fornisce come risultato un valore di tipo funzione;
- Un modo per associare nomi a valori nell'ambiente globale (`define` in scheme, `val` in Standard ML, etc...)
- Dipendentemente dal sistema di tipi usato dal linguaggio, vengono forniti modi per definire nuovi tipi di dato, per combinare tipi esistenti tramite prodotto cartesiano (tuple) o unione, etc...

Il modo in cui un linguaggio di programmazione implementa questi punti da' origine a differenti linguaggi di programmazione funzionale, con caratteristiche diverse. Per esempio, il sistema di tipi utilizzato dal linguaggio può variare, facendo controlli più o meno stretti, a tempo di esecuzione (come per esempio in Lisp, scheme e linguaggi simili) o a tempo di compilazione (come per esempio in Haskell, Standard ML, ma anche C++). Un sistema di tipi più dinamico (e meno "stretto") aumenterà la possibilità di commettere errori di programmazione (non riuscendo ad identificare alcune classi di errori legati ai tipi) ed introdurrà un maggiore overhead (effettuando alcuni controlli a runtime invece che a tempo di compilazione), ma d'altra parte risulterà più flessibile e potente (per esempio, implementare un Y combinator in Lisp o scheme è molto più facile che implementarlo in Haskell, Standard ML o C++).

Per quanto riguarda invece la definizione di funzioni, alcuni lettori saranno probabilmente più abituati all'approccio seguito da molti linguaggi di programmazione imperativi, che generalmente forniscano un unico meccanismo che contemporaneamente specifica il corpo della funzione e crea nell'ambiente un legame fra il corpo della funzione ed il suo nome. Al contrario, i più comuni linguaggi di programmazione funzionali distinguono due diversi meccanismi: l'astrazione, che genera un valore di tipo funzione senza assegnargli un nome (una cosiddetta "funzione anonima" — si consideri come esempio il costrutto lambda expression del C++, accennato in precedenza) ed un secondo meccanismo che permette di modificare l'ambiente (anche globable) associando un nome ad un generico valore (che può essere di tipo funzione o altro). Come importante conseguenza, *nel momento in cui si definisce il corpo di una funzione tale funzione non è ancora associata ad un nome*. Questo può chiaramente creare dei contrattempi quando si cerca di definire una funzione ricorsiva, come si vedrà meglio in futuro parlando di $\lambda$ calcolo.

Per finire, un aspetto fondamentale dei linguaggi di programmazione funzionale è chiaramente l'invocazione di funzione. Sebbene possa sembrarci una cosa semplice e naturale, merita un minimo di discussione: se ci troviamo infatti di fronte alla definizione della funzione `fattoriale(unsigned int n)` definita come "(n == 0) ? 1 : n * fattoriale (n − 1)", ci viene infatti naturale pensare che " fattoriale (4)" sia valutata come

(4 == 0) ? 1 : 4 * fattoriale (4 − 1) → 4 * fattoriale (3) → ...

andando in pratica ad eseguire subito l'operazione $4 - 1 = 3$, ma questa non è l'unica soluzione possibile. Un'alternativa potrebbe essere

(4 == 0) ? 1 : 4 * fattoriale (4 − 1) → 4 * fattoriale (4 − 1) →
→ 4 * ((4 − 1 == 0) ? 1 : (4 − 1) * fattoriale (4 − 1 − 1)) →
→ 4 * ((4 − 1) * fattoriale (4 − 1 − 1)) → ...

eseguendo in pratica le operazioni aritmetiche solo quando strettamente necessario.

Un linguaggio di programmazione funzionale deve quindi specificare in modo chiaro come e quando valutare le espressioni. In particolare, si possono avere:

- strategie di valutazione *eager*, in cui quando una funzione `f` è applicata ad un'espressione `e` l'espressione è valutatata (riducendola ad un valore irriducibile) prima di invocare la funzione
- strategie di valutazione *lazy*, in cui quando una funzione `f` è applicata ad un'espressione `e` la funzione è invocata senza prima provare a ridurre l'espressione che riceve come argomento (passando quindi un'espressione non valutata e non un valore irriducibile).

Si noti come la prima strategia coincide sostanzialmente con il passaggio di parametri per valore (il parametro attuale è valutato fino a divenire un valore irriducibile prima di invocare la funzione), mentre la seconda strategia coincide col passaggio di parametri per nome (alla funzione viene passato un *thunk*, vale a dire un'espressione senza variabili locali, che non viene valutata prima di essere effettivamente utilizzata).

Generalmente, i linguaggi funzionali puri (come Haskell) tendono a prediligere valutazioni lazy, mentre linguaggi funzionali che ammettono effetti collaterali (come ML, Scheme, etc...) sono costretti ad usare strategie di valutazione eager. Per capire come mai, si consideri un linguaggio con variabili modificabili (e quindi "non troppo funzionale") ed una funzione `bad_bad_function()` definita come

```
void bad_bad_function(void)
{
  x++;
}
```

dove "`x`" è una variabile globale. Se il valore di "`x`" è inizialmente 0, quanto vale "`x`" dopo aver invocato some_function(bad_bad_function(), bad_bad_function())? Se si usa una strategia di valutazione lazy, non è possibile dirlo a priori, perché dipende da quante volte `some_function()` valuta i suoi argomenti (mentre se si usa una strategia eager è possibile dire che `bad_bad_function()` verrà invocata una volta per argomento e quindi il valore di "`x`" sarà 2).

Sebbene il precedente esempio riguardi sostanzialmente un non-problema (i linguaggi di programmazione funzionale non dovrebbero implementare variabili modificabili), le operazioni di I/O rappresentano problemi ben più reali e seri. Qualsiasi operazione di input o output costituisce infatti un effetto collaterale ed in presenza di valutazione lazy crea quindi dei non-determinismi nel comportamento del programma (quale sarebbe l'output di some_function(bad_bad_function(), bad_bad_function()) se `bad_bad_function()` stampasse qualcosa sullo schermo?). I linguaggi funzionali puri generalmente affrontano questo problema modellando le funzioni di I/O come funzioni che ricevono un'entità "mondo" in ingresso e producono in output una versione modificata di tale "mondo". I linguaggi che prevedono valutazione lazy forniscono poi vari tipi di meccanismi per serializzare l'esecuzione delle operazioni di I/O, in modo da rendere deterministiche le interazioni del programma col mondo esterno. Tipicamente, la serializzazione dell'esecuzione di due funzioni `f1()` ed `f2()` è implementabile facendo si che `f2()` sia invocata ricevendo in ingresso l'output di `f1()` (l'entità "mondo", per esempio). Poiché questa soluzione porta a notazioni complesse e poco intuitive, alcuni linguaggi come Haskell forniscono una sintassi semplificata per questi meccanismi, che ricorda la sintassi dei linguaggi imperativi (per fare questo, Haskell utilizza strumenti matematici complessi come le monadi della teoria delle categorie).

Gli effetti pratici dell'uso di differenti strategie di valutazione sono visibili, per esempio, provando ad implementare il Y combinator (un'implementazione in Haskell non darà problemi, mentre un'implementazione in Standard ML o Scheme non sarà possibile e costringerà ad implementare un differente combinator, come per esempio Z).

E' dimostrabile che se il meccanismo di valutazione lazy e quello eager riducono entrambi un'espressione ad un valore, allora il valore ottenuto tramite valutazione lazy e quello ottenuto tramite valutazione eager coincidono (a tale proposito, vedere il teorema di Church-Rosser). Inoltre, se la valutazione lazy porta ad una ricorsione infinita allora anche la valutazione eager porta ad una ricorsione infinita (ma d'altra parte esistono situazioni in cui la valutazione eager genera ricorsione infinita mentre la valutazione lazy permette di ridurre l'espressione ad un valore - ancora, vedere Y combinator).

# 6 Computazione per Riduzione

Basandosi sui meccanismi appena descritti, un programma scritto secondo il paradigma funzionale può essere eseguito tramite la computazione per riduzione, implementata ripetendo 2 operazioni:

- Ricerca di nomi nell'ambiente (e sostituzione testuale di un nome con il corrispondente valore funzionale - rappresentato come astrazione)
- Applicazione di funzioni (sostituzione testuale del parametro formale col parametro attuale)

Un programma funzionale è quindi rappresentato come un insieme di definizioni ed operazioni di modifica dell'ambiente (creazioni di binding) che per essere processate possono richiedere la valutazione di espressioni. La computazione di tale programma verrà effettuata dalla macchina astratta tramite una serie di riscritture / riduzioni che porteranno a semplificare fino a che non si arriva a forme semplici non ulteriormente riducibili (dette *valori*).

Per rendere meno ostico l'utilizzo di tecniche di programmazione funzionale vengono spesso forniti altri costrutti che pur non essendo strettamente necessari semplificano notevolemente lo sviluppo del codice. Esempi sono:

- Un modo per modificare l'ambiente locale (generalmente, il costrutto `let`);
- Un meccanismo analogo al fixed point operator `fix`, che permette di definire funzioni ricorsive;
- Alcuni costrutti che costituiscono uno "zucchero sintattico" per definire funzioni a più argomenti (nascondendo l'utilizzo dell currying), etc...

Rigurardo al costrutto (generalmente chiamato `let`) usato per modificare l'ambiente locale in cui viene valutata un'espressione, si noti che questo costrutto non è strettamente necessario perché implementabile tramite chiamata a funzione e passaggio parametri. Per esempio, si consideri un generico costrutto "`let` *x* `=` *e1* `in` *e2*" (dove "*x*" è un generico nome mentre "*e1*" e "*e2*" sono due espressioni) che associa il nome "*x*" all'espressione "*e1*" durante la valutazione di "*e2*". Questo è implementabile definendo una funzione `f()` con parametro formale "*x*" e corpo "*e2*" ed invocando tale funzione con parametro attuale "*e1*"[7]. Ancora meglio, si può usare una funzione anonima: usando la sintassi di Standard ML "`let` *x* `=` *e1* `in` *e2*" diventa "(**fn** x => e2) e1".

Il costrutto equivalente al fixed point operator `fix` è invece molto utile per semplificare la definizione di funzioni ricorsive: come accennato in precedenza (e come diventerà più chiaro studiando il $\lambda$ calcolo), senza questo meccanismo la definizione di funzioni ricorsive non sarebbe possibile in modo semplice: il nome di una funzione non può essere usato nella sua definizione, perché non è ancora legato a nessun valore nell'ambiente globale. Per definire funzioni ricorsive sarebbe necessario implementare un fixed point combinator (come Y o Z) ed applicare tale operatore alla "versione chiusa" della funzione che si vuole definire. Questo meccanismo (chiamato `val rec` o `fun` in Standard ML, `letrec` in Scheme, etc...) permette invece di usare la ricorsione in modo diretto.

In linguaggi che usano sistemi di tipi più potenti è spesso fornito anche un meccanismo di *pattern matching* che permette di manipolare valori di tipi definiti dall'utente (per esempio, distinguendo i vari varianti di un tipo, etc...).

# 7 Un Linuguaggio Funzionale Minimale

Per concludere, i lettori più curiosi potrebbero chiedersi come sia fatto il più semplice linguaggio di programmazione funzionale possibile, che non contenga funzionalità "di alto livello" utili per rendere il codice più leggibile ma non strettamente necessarie. In altre parole, cosa si ottiene rimuovendo da un linguaggio di programmazione funzionale le caratteristiche non indispensabili per la Turing-completezza, come

- la presenza di un ambiente globale (che come detto semplifica la definizione di funzioni ricorsive)
- la "tipizzazione stretta' e la presenza di tipi di dati più complessi (che aumentano la leggibilità del codice ma non sono fondamentali: si noti come anche nel paradigma di programmazione imperativo il linguaggio Assembly non definisca tipi di dato ma consideri solo valori binari)
- i vari costrutti che costituiscono "zucchero sintattico".

Quel che resta è un linguaggio in cui i programmi sono espressioni (pure!) composte semplicemente da:

1. nomi (termini irriducibili)

[7] Questo trucco di sostituire una variabile con un parametro formale ed il suo valore col parametro attuale è spesso usato per reimplementare codice imperativo usando il paradigma funzionale.

2. definizioni di funzioni (il concetto di astrazione)

3. applicazioni di funzioni

Per quanto riguarda i nomi, la scelta più semplice e minimale è quella di usare singole lettere minuscole, anche se talvolta si permettono di usare identificatori composti da più caratteri.

Per quanto riguarda l'applicazione di funzioni, i programmatori che hanno familiarità con linguaggi della famiglia del C (C, C++, Java, ...) sono abituati ad indicare con "`f(x)`" l'applicazione della funzione "`f`" al parametro attuale "`x`" (si ricordi che per semplicità si possono considerare solo funzioni ad un argomento). Le parentesi attorno al parametro attuale sono però inutili, quindi si potrebbe anche usare la sintassi "`f x`", che è spesso preferita. La se l'applicazione di funzioni associa a sinistra, la composizione $g \circ f$ delle funzioni $f$ e $g$ può essere quindi implementata come "`g (f x)`" invece che "`g(f(x))`". La sintassi "`g f x` è invece equivalente a "`(g(f))(x)`" (e questo, come si vedrà, rende più naturale la sintassi del currying). Alcuni linguaggi della famiglia LISP prevedono invece le parentesi attorno all'applicazione di funzione invece che attorno al parametro attuale ("`(f x)`" invece che "`f(x)`"); in questo caso, $g \circ f$ diventa "`(g (f x))`".

Per finire, il linguaggio deve prevedere di costruire espressioni che vengono valutate a funzioni (permettendo in qualche modo di "definire" una funzione a partire da un'espressione "`e`" ed un parametro formale "`x`"). Indipendentemente dalla sintassi che il linguaggio usa, questo costrutto *astrae* l'espressione "`e`" dallo specifico valore del parametro formale "`x`"; deve quindi contenere una qualche keyword specifica del linguaggio (che può essere la lettera greca $\lambda$, il simbolo "\", la parola "`fn`", una sequenza di parentesi quadre, tonde e graffe, o altro), il nome del parametro formale e l'espressione da astrarre. Esempi possono essere "$\lambda x.e$", "`\x -> e`", "`fn x => e`", "`[](auto x) { e }`", "`(lambda (x) (e))`" o simili...

Non prevedendo una tipizzazione stretta, questo "linguaggio minimale" conosce solo generiche "funzioni" che operano su espressioni (ricevono un'altra funzione generica come argomento e generano una funzione generica come risultato), senza che siano specificati in modo più preciso dominio e codominio di tali funzioni (tali insiemi coincidono con l'insieme delle espressioni che compongono il linguaggio). Ma sorprendentemente il linguaggio risultante (noto come $\lambda$ calcolo) è ancora Turing completo: è possibile codificare valori naturali, booleani e di altro tipo usando solo funzioni ed è possibile usare vari tipi di fixed point combinator per implementare funzioni ricorsive anche in assenza di ambiente non locale. Per questo motivo, il $\lambda$ calcolo è spesso considerato come una sorta di "Assembly dei linguaggi di programmazione funzionale".