Strutturare il Controllo (della valutazione)

Sommario: 22 marzo, 2016

- Controllo della sequenza: Linguaggi Machine-Level, High-level e Valori modificabili
- Espressioni: Valutazione e Controllo della sequenza
- Comandi vs. Espressioni: Assegnamento
- Comandi: Controllo della sequenza
- Comandi Strutturati: goto e comandi per iterazione
- Programmazione Strutturata: Antesignana delle moderne Metodologie
- Ricorsione: Programmazione con induzione, Tail Recursion e Memoization

Controllo della sequenza: Linguaggi Machine-Level, e Valori Modificabili

- **Linguaggi Machine-Level**:
  - La struttura di un programma è una sequenza di statement atomici:

```
Mov L0, R0
Add R1, R0
Mov R1, L0
```

    - che usano Locazioni come Valori Modificabili
  - La sequenza di valutazione è facilmente controllata dal registro PC.
  - questa sequenza è determinante quando sono coinvolti Valori Modificabili

- Linguaggi High-Level:

Controllo della sequenza: Linguaggi High-level e Valori Modificabili

- Linguaggi Machine-Level:
- **Linguaggi High-Level**:
  - La struttura di un programma può non essere una seq.
  - Gli statements possono non essere atomici
    - ma possono usare Variabili come Valori Modif.
  - La sequenza di valutazione dipende dalla semantica.
- Semantica e Controllo di sequenza.

Osservazione (Translation Semantics)

La Semantica ci dice:
Come stms non atomici sono decomposti in stm più semplici e in quale sequenza questi ultimi devono essere considerati e valutati

Controllo della sequenza: Espressioni

Espressioni sono presenti in tutti gli High-Level L.

- Semantica: In tutti i linguaggi, la sem. prevede che calcolino un valore, o risultino in una computazione indefinita (non-terminante).
- Struttura:
  - Un operatore (principale) ed $\texttt{n} \geq 0$ argomenti (a loro volta espressioni)
  - Un'invocazione di funzione applicata a $\texttt{n} \geq 0$ argomenti (a loro volta espressioni)
  - Un termine atomico (variabile, valore atomico)
- Controllo di sequenza: segue l'ordine e il metodo di valutazione degli argomenti
  - Ordine: leftmost, rightmost, associatività dx o sin.
  - Metodo:
    - Operatori: stretto, circuito-corto,
    - Invocazioni: trasmissione parametri

Espressioni: Ordine e Metodo di valutazione

Espressioni sono presenti in tutti gli High-Level L.

- **Ordine** di valutazione: *leftmost*, *rightmost*, *assoc. dx o sin.*
  - critico quando:
    - operatori su aritmetica finita (virgola mobile)
    - effetti laterali (modifica di variabili):
      Es. (x=3)+(y=x)
- **Metodo** di valutazione: *stretto*, *circuito-corto*
  - critico quando:
    - argomento indefinito.
      Esempio. Sia:
      - "e" valuta indefinito. Allora:
        $(\lambda\, \mathtt{x}.\lambda\, \mathtt{y}.\mathbf{if}\ (\mathtt{x} > 0)\ \mathbf{then}\ \mathtt{x}\ \mathbf{else}\ \mathtt{x} + \mathtt{y})15\ \mathtt{e}$
        calcola 15 se non stretta sul primo argomento
      - "e_1" o "e_2" valuta indefinito e l'altra valuta **false**.
        e_1 **Or** e_2
        calcola **false** se **Or** è valutata a circuito-corto

Assegnamento: Espressione o Comando

Assegnamento

- è un'espressione in alcuni linguaggi (C, Java).
- è un comando in altri (Pascal).
- Semantica:
  - In tutti i casi modica un valore modificabile
  - Linguaggi differenti possono usare differenti ordini di valutazione degli argomenti.
  - conducendo a sequenze di valutazione completamente diverse e computazioni completamente diverse

```
b = 0;
a[f(b)] = a[f(b)]+1
```

per f così definito (e non locale "b" quella dell'assegnamento sopra):

```
int f(int x){
  if (x==0){b=1; return 1;}
  else return 2;
}
```

  - usare ora ordine leftmost ora rightmost.

Assegnamento: l-value e r-value

Assegnamento:

- Gli argomenti di un assegnamento devono essere un valore modificabile (l-value) e un valore memorizzabile (r-value)
  - valore modificabile = Locazione di memoria
  - valore memorizzabile = valore assegnabile ad una locazione nella memoria
  - valore denotabile = valore associabile ad un identificatore nell'ambiente
  - valore esprimibile = valore che può essere calcolato con un'espressione (non singola variabile)
- Alcune espressioni possono essere valutate per ottenere un l-value oppure un r-value

```
a[f(b)] = a[f(b)] +1
```

Controllo della sequenza: Comandi

Comandi sono presenti in tutti i Linguaggi Prescrittivi (Macchina, Imperativi,...)

- Semantica: In tutti i linguaggi, la semantica prevede che modifichino la memoria (effetti laterali).
- Struttura. Varie classi di comandi:
  - Controllo di sequenza esplicito
  - Condizionali (o di selezione)
  - Iterativi

Comandi di Sequenza Esplicito

Introducono una sequenza di C, o alterano la sequenza corrente.

- Includono:
  - Comando Sequenza: $\texttt{C}_1\texttt{;}\,\texttt{C}_2$
  - Blocco inline
  - goto, break, continue, return.
- Comando Sequenza e Blocco inline sono presenti in tutti i Linguaggi con comandi ed hanno una semantica composizionale ottenuta dalla naturale composizione della semantica dei componenti.
- goto: Iterazione non strutturata
  - oscuro e poco espressivo
  - può sempre essere evitato a favore di iterazione strutturata
  - va evitato
- break, continue, return
  - Utilizzati in C per uscita (non strutturata??) da blocchi.

Comandi di Condizionali

Discriminano tra due o più sequenze alternative

- Includono:
  - if-then-else e if-then
    - con struttura e comportamento analogo in tutti i linguaggi
  - case e switch
    - con struttura simile ma comportamento, a volte, diverso

      $\texttt{case 2 of 1 : C}_1\texttt{; 2 : C}_2\texttt{; 3 : C}_3\texttt{; 4 : C}_4\texttt{; end; C;}$

      $\texttt{switch (2)\{case 1 : C}_1\texttt{; case 2 : C}_2\texttt{; case 3 : C}_3\texttt{; case 4 : C}_4\texttt{; \}C;}$

      conducono alla sequenza:

      $\texttt{C}_2\texttt{; C;}$

      $\texttt{C}_2\texttt{; C}_3\texttt{; C}_4\texttt{; C;}$

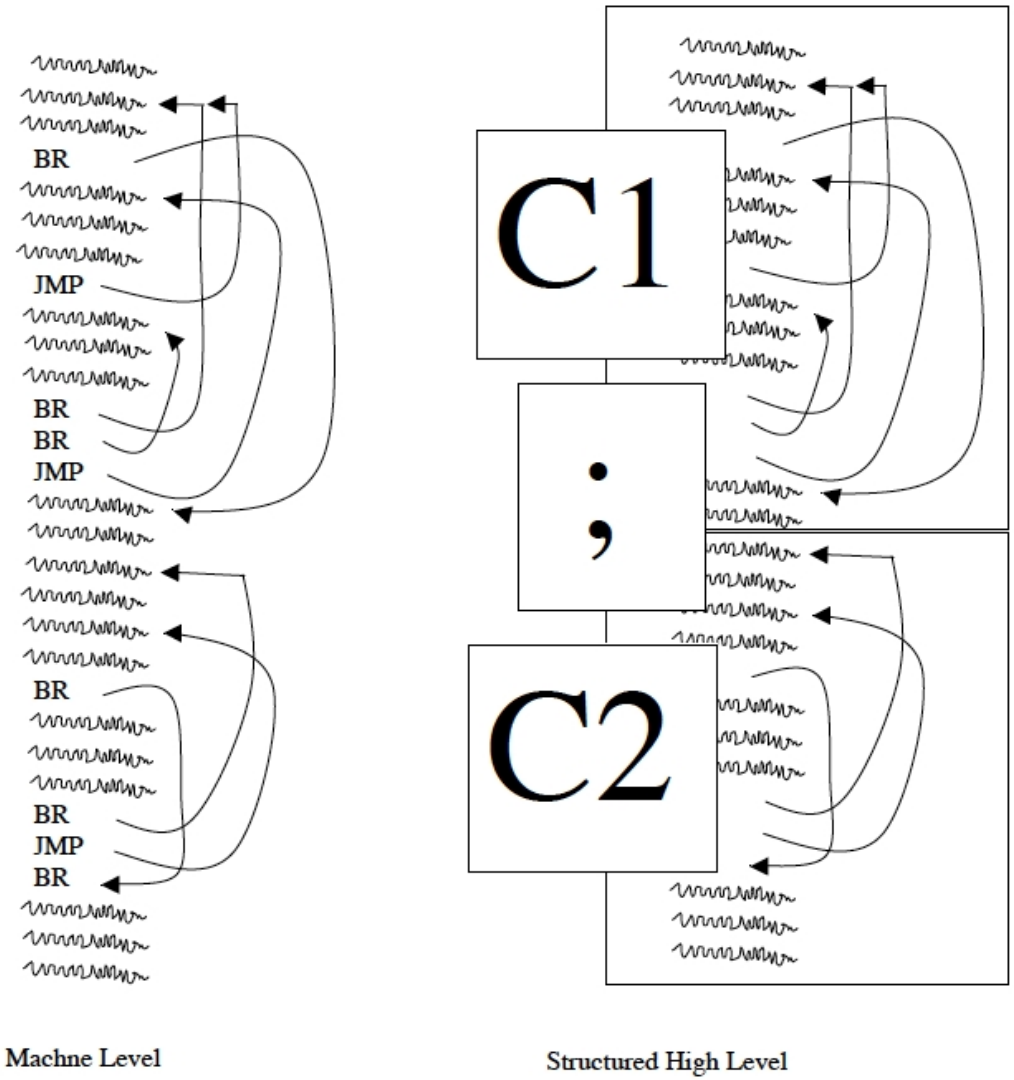
      rispettivamente.

Comandi Iterativi (strutturati)

Incapsulano ed evidenziano la sequenza da iterare

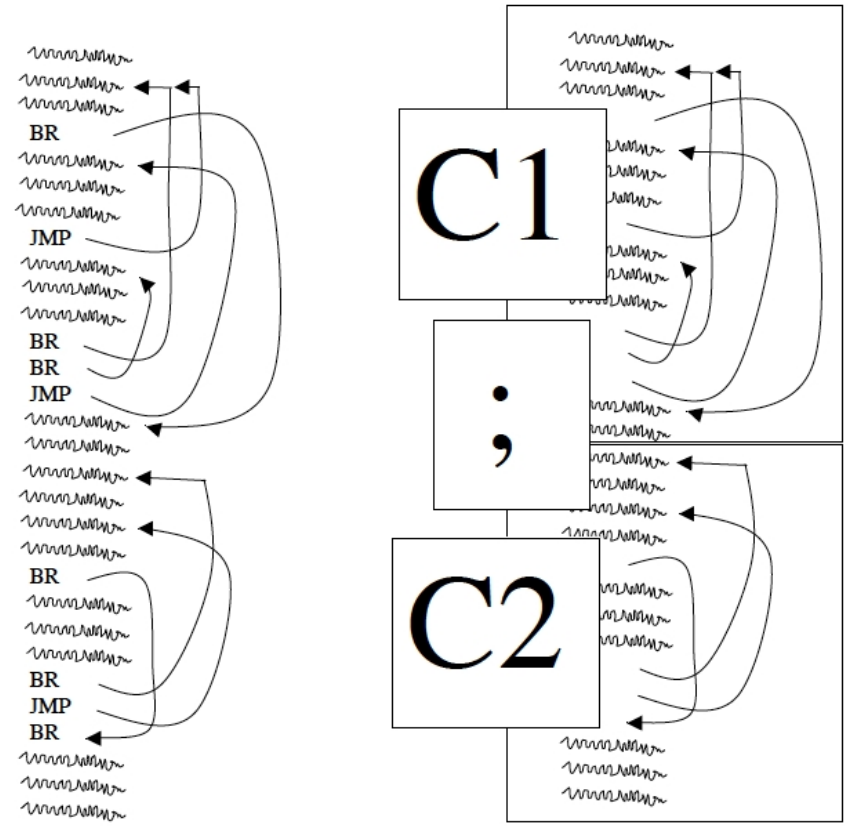
- Si differenziano in questo dall'iterazione non strutturata basata sul goto
- Si suddividono in:
  - iterazione indeterminata: while, repeat, for (del C)
  - iterazione determinata: for del Pascal
- Iterazione indeterminata: **while** `E` **do** `C`
  - il numero di iterazioni può non essere determinato utilizzando lo stato iniziale e l'espressione di controllo `E`
  - l'iterazione può anche non terminare
  - `while (x>1 and x<10) do x = 5;`
- Iterazione determinata: **for** `i` $=$ $\mathtt{E}_{\mathtt{inizio}}$ **to** $\mathtt{E}_{\mathtt{fine}}$ **do** `C`
  - il numero di iterazioni è: $\mathtt{n} = \mathtt{v}_{\mathtt{E}_{\mathtt{fine}}} - \mathtt{v}_{\mathtt{E}_{\mathtt{inizio}}}$

Costrutto Strutturato: Sequenze chiuse



- Possiamo sempre "estendere" una sequenza atomica con JMP e BR in una sequenza atomica equivalente formata di sole sezioni "chiuse"

Costrutto e Programma Strutturato



- Costrutto Strutturato: un costrutto non atomico che definisce una sequenza atomica "chiusa", ovvero ha un unico punto di ingresso (il primo atomo) ed un unico punto di uscita (l'ultimo atomo).
- Linguaggio, Programma Strutturato: Le funzioni calcolabili possono essere espresse con programmi che utilizzano solo costrutti strutturati (Bohm-Jacopini, 1966)

Costrutti non strutturati

- Costrutto Strutturato: un costrutto non atomico che definisce una sequenza atomica "chiusa", ovvero ha ha un unico punto di ingresso (il primo atomo) ed un unico punto di uscita (l'ultimo atomo).

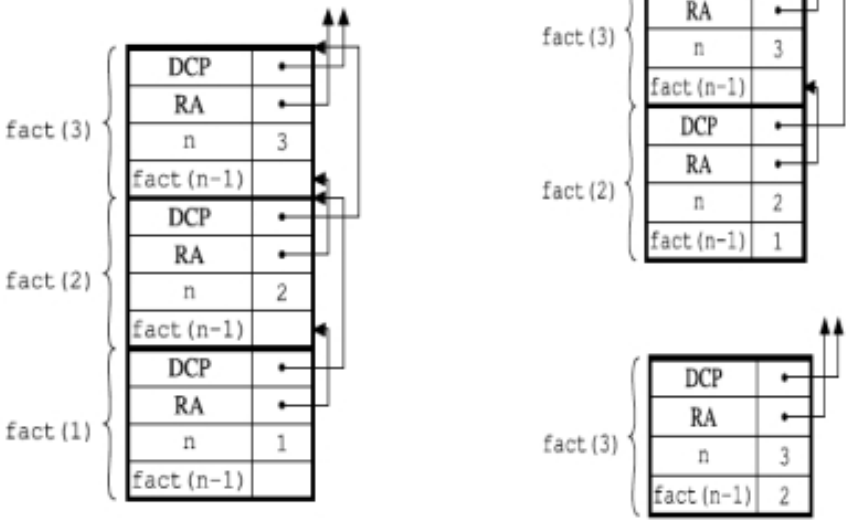
- Linguaggio/Programma Strutturato: Le funzioni calcolabili possono essere espresse con programmi che utilizzano solo costrutti strutturati (Bohm-Jacopini, 1966)

- Rispondiamo:
  - Quali dei seguenti costrutti sono strutturati e perchè?
    (a) Goto; (b) break; (c) continue; (d) return
  - Il costrutto Switch del C è un costrutto strutturato?

Ricorsione

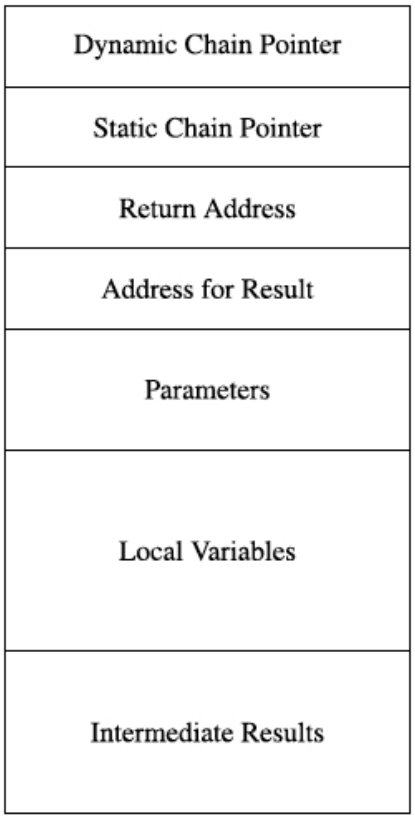
- Algoritmi definiti in modo induttivo
- Introdotta attraverso Procedure/Funzioni
- Possono richiedere l'allocazione di una grande quantità di AR

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



- Dire chi sono i campi: DCP, RA presenti negli AR e di quale campo fa parte l'entry "fact(n-1)..."

AR per Memoria a Stack



- Ricordiamo la struttura generale vista

Ricorsione: Funzioni Parziali e Gestione delle Eccezioni

- Possiamo ridurre l'allocazione di AR al solo AR dell'invocazione corrente
- Ricorsione di coda: È una tecnica che lo permette
- Prima però commentiamo la definizione data per `fact`.

```
int fact(int n){** fact calcola 0 se n è negativo, altrimenti n! **
        if (n<0) return 0;
          else if (n==0) return 1;
            else return n * fact(n - 1);
}
```

  - C non ha un meccanismo di eccezioni che sarebbe stato preferibile a questo uso arbitrario (e oscuro) del valore calcolato 0

```
bool fact(int n, int * r){** Convenzione in C per le parziali**
        if (n<0) return false;
          else if (n==0){*r = 1; return true; }
              else{*r = n * fact(n - 1); return true; }
}
```

  - Il commento inserito è d'obbligo ma non risolve il problema

Ricorsione: Funzioni Parziali e Gestione delle Eccezioni - 2

- Il commento inserito è d'obbligo ma non risolve il problema delle parziali

```
int fact(int n){** fact calcola 0 se n è negativo, altrimenti n! **
        if (n<0) return 0;
          else if (n==0) return 1;
            else return n * fact(n - 1);
    }

bool fact(int n, int  * r){** Convenzione in C per le parziali**
       if (n<0) return false;
         else if (n==0){*r = 1; return true; }
              else{*r = n * fact(n - 1); return true; }
  }
```

- la definizione sotto è una corretta definizione della funzione **parziale** n!.

```
int fact(int n){** fact calcola 0 se n è negativo, altrimenti n! **
        if (n==0) return 1;
        else return n * fact(n - 1);
    }
```

  - Noi vogliamo usare proprio quella parziale nei nostri programmi?
  - Nella quasi totalità dei casi NO
  - Esprimere una definizione che ci dica quando le cose non vanno:
    senza "falsare" il valore calcolato
    ovvero, faccia uso di un *meccanismo di eccezioni*.

Ricorsione: Ricorsione di Coda

- Possiamo ridurre l'allocazione di AR al solo AR dell'invocazione corrente
- Ricorsione di coda: È una tecnica che lo permette

```
int fact(int n){** fact calcola 0 se n è negativo, altrimenti n! **
        if (n<0) return 0;
          else if (n==0) return 1;
              else return n * fact(n - 1);
  }
```

Definition (Invocazione di Coda e Ricorsione di coda)

Sia `F` una definizione di funzione che contiene un'invocazione di una funzione `g`. Tale invocazione è detta invocazione di coda se, quando applicata, `F` restituisce il valore calcolato da tale invocazione senza ulteriore calcolo. Una definizione ricorsiva `F` è con ricorsione di coda se ogni sua invocazione contenuta in `F` é una invocazione di coda.

- la definizione di `fact` non è con ricorsione di coda
- questa sotto lo è

```
int factT(int n, int acc){** acc produttoria argomenti n di invocazioni precedenti **
          if (n<0) return 0;
            else if (n==0) return acc;
                else return fact(n - 1, n * acc);
   }
```

- Per ogni intero `n`, `factT(n,1)` calcola `n!`.

Ricorsione: Macchine Astratte per Ricorsione di Coda

AR nell'invocazione di funzione con Ricorsione di Coda.

- Sia $g(e_0)$ l'invocazione di una tale funzione, con e_0 (lista degli) argomenti dell'applicazione.
- L'invocazione può condurre a due casi possibili:
  - Calcola un valore senza richiedere ulteriori invocazioni di g. Caso finale
  - Conduce ad un'invocazione di coda $g(e_1)$. Allora $g(e_1)$ è tutto ciò che serve per calcolare e_0
    - Possiamo rimpiazzare l'invocazione $g(e_0)$ con $g(e_1)$, ovvero
    - Possiamo, rimpiazzare il contenuto dell'AR prodotto per calcolare $g(e_0)$ con quello per $g(e_1)$

Ricorsione di Coda: Activation Record di factT

- Sia g(e_0) l'invocazione di ... con e_0 (lista degli) argomenti dell'applicazione.
- L'invocazione può condurre a due casi possibili:
  - Calcola un valore senza richiedere ulteriori invocazioni di g. Caso finale
  - Conduce ad un'invocazione di coda g(e_1). Allora g(e_1) è tutto ciò che serve per calcolare e_0
    - Possiamo rimpiazzare l'invocazione g(e_0) con g(e_1), ovvero
    - Possiamo, rimpiazzare il contenuto dell'AR prodotto per calcolare g(e_0) con quello per g(e_1)

Applichiamolo a factT(3,1), definita sotto

```
int factT(int n, int acc){** acc produttoria argomenti n di invocazioni precedenti **
          if (n<0) return 0;
            else if (n==0) return acc;
                 else return fact(n − 1, n ∗ acc);
    }
```

...	...
CD	...
RA	...
n	3
acc	1
fact(n-1,n*acc)	

...	...
CD	...
RA	...
n	2
acc	3
fact(n-1,n*acc)	

...	...
CD	...
RA	...
n	1
acc	6
fact(n-1,n*acc)	

Ricorsione: Memoization e Memoria Statica nelle Procedure

- Possiamo ridurre l'allocazione di AR al solo AR dell'invocazione corrente
- Ricorsione Memoized: È una tecnica che lo permette *parzialmente*

Definition (Funzione Memoized)

Una definizione di funzione è Memoized se memorizza tutte le coppie (invocazione, valore-calcolato) durante l'intera esecuzione del programma, fornendo il valore calcolato memorizzato se già invocata sull'argomento, calcolando il nuovo valore altrimenti.

```
int fact(int n){/* diverge quando n<0, altrimenti n! */
    if (n==0) return 1;
    return n * fact(n-1);
}
int memoizedFact(int n){/* diverge quando n<0, altrimenti n! */
                        /* definizione memoized */
    static int Tab[100];
    static int index=0;
    Tab[0]=1;
    if ((n<=index)&!(n<0)) return Tab[n];
    Tab[n] = n * memoizedFact(n-1);
    return Tab[index=n];
}
```

- la definizione di `memoizedFact` è con Memoized
- la definizione utilizza Memoria Statica nell'Activation Record di `memoizedFact`

Ricorsione: Ricorsione di Coda, Memoization

- Due tecniche completamente diverse
- Ricorsione di Coda richiede:
  - Una definizione con ricorsione di coda (fornito da utente);
  - Un meccanismo per riuso dell'AR (fornito dal Linguaggio);
  - Presente nei linguaggi tipo Scheme, Miranda (tutti funzionali);
- Memoization richiede:
  - Un meccanismo per ricordare le invocazioni (Tabella hash);
  - C non lo possiede e noi lo abbiamo emulato nell'esempio;
  - La nostra emulazione ha usato Memoria Statica,
  - Ma potevamo usare anche Memoria Dinamica,
  - ed anche Permanente (un file del File System).
  - Presente in Common Lisp, Perl, Python (manipolazione simbolica)

- Concludiamo:
  - Ricorsione di coda applicabile solo a funzioni definite ricorsive e utilizzabile solo in Linguaggi dotati di meccanismo per il riuso dell'AR.
  - Memoization utilizzabile in tutti i linguaggi e applicabile a tutte le funzioni, richiede che non siano presenti effetti laterali.

Programmazione Strutturata/1

Antesignana delle moderne metodologie di sviluppo

- Progettazione top-down del programma:
  - Una Specifica Astratta iniziale che mostra funzionalità attese e correlazioni tra esse
  - raffinamenti successivi che aggiungono dettagli e ripetono il processo su ciascuna funzionalità
- Modularizzazione del codice:
  - Ogni codice che implementa una funzionaltà deve essere localizzato in una parte precisa del programma
  - che deve essere acceduta e trattata come un unità di programmazione indivisibile (un modulo)
- Uso di identificatori significativi:
  - Identificatori adatti a indicare ruolo ed agevolare la comprensione del codice dove sono usati
- Uso estensivo di commenti
  - Commenti adatti a fornire indicazioni utili su vincoli ed utili per il testing, la modifica e la manutenzione del codice

Programmazione Strutturata/2

Antesignana delle moderne metodologie di sviluppo

- Progettazione top-down del programma:
- Modularizzazione del codice:
- Uso di identificatori significativi:
- Uso estensivo di commenti
- Uso di Tipi di dato strutturato
  - adatti a rappresentare in modo unitario valori aventi strutture anche complicate
- Uso di costrutti strutturati che devono:
  - mostrare in modo chiaro la trasformazione calcolata.

Progr. Strutturata: Applichiamola al Quicksort /1

La specifica astratta iniziale può coincidere con la descrizione del problema, vincoli e soluzione e/o algoritmo.
Ad esempio:

- Obiettivo Generale: Programma di Ordinamento
- Obiettivo Specifico: Quicksort per sequenze ordinabili
- Vincoli:
  - Una sequenza **c**
  - di valori ordinabili **t**
  - Quindi, **c** è un valore di tipo **Seq(t)**, sequenze di tipo `t`.
- Soluzione/Algoritmo:
  a. Sia **cc** la sequenza corrente da ordinare
  b. Sia **a** di tipo **t**, un elemento di separazione per **cc**
  c. Dividiamo **cc** in due sottosequenze **cLE** e **cGT** separate da **a**
  d. Ripetere [a]-[d] su ogni sottosequenza non singoletta, ottenuta al passo [c] se presente.
  e. Terminiamo, presentando la sequenza ottenuta.

Progr. Strutturata: Applichiamola al Quicksort /2

Procediamo con `N` raffinamenti successivisi, individuando funzionalità e aggiungendo dettagli

- Hints: Chiarire e precisare dettagli rispondendo a domande, quali:
- Cosa si deve intendere per elemento di separazione?
- Cosa si deve intendere per divisione di sequenza rispetto a elemento di separazione, dato un ordinamento?
- ...