Strutturare il Controllo di Sequenza

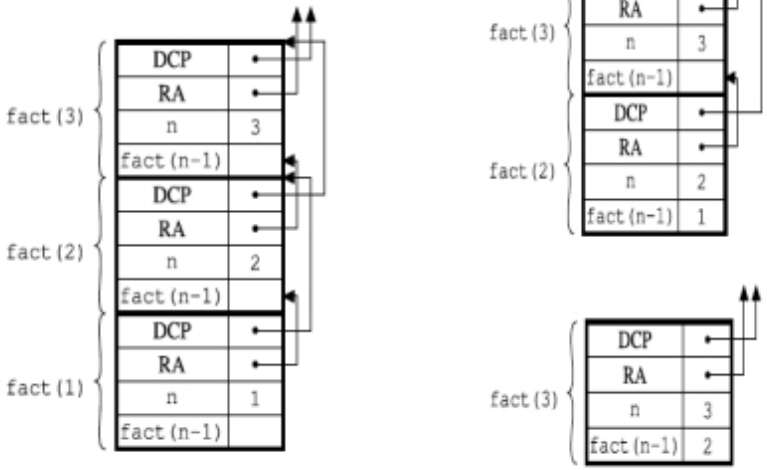
Sommario: 21 marzo, 2019

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

Ricorsione

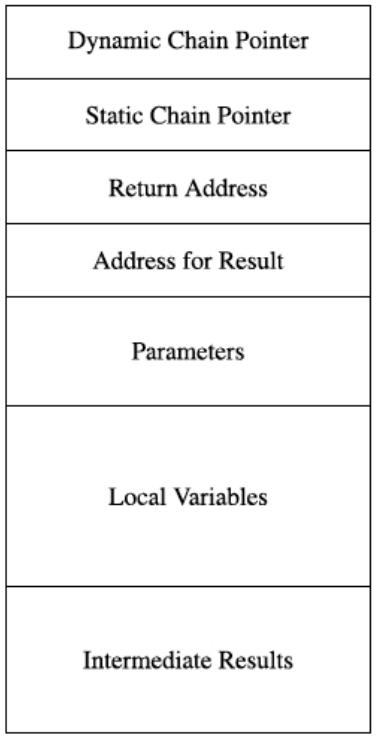
- Algoritmi definiti in modo induttivo
- Introdotta attraverso Procedure/Funzioni
- Possono richiedere l'allocazione di una grande quantità di AR

```
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;
        if (n==0) return 1;
        return n * fact(n-1);
}
```

  - C non ha un meccanismo di eccezioni che sarebbe stato preferibile a questo uso arbitrario (e oscuro) del valore booleano calcolato, `false`.

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

  - 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;
        if (n==0) return 1;
        return n * fact(n-1);
}

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

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

```
int fact(int n){** fact diverge se n è negativo, altrimenti n! **
        if (n==0) return 1;
        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 delle invocazioni precedenti **
          if (n<0) return 0;
            else if (n==0) return acc;
               else return factT(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 g su 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 calcolare $g(e_1)$

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 g su 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 calcolare g(e_1)

Applichiamolo a factT(3,1), definita sotto

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

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

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

...	...
CD	...
RA	...
n	1
acc	6
factT(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 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 ed è 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 procedimento 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 utilizzabili per documentazione, testing, modifica e 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

- Sequenze con ripetizione: Gli elementi possono essere ripetuti?
- **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?
  - Come può essere fornito/definito un ordinamento per i valori da ordinare?
  - ...

Esercizi

1. Perchè il Controllo di Sequenza è un insieme di meccanismi presenti solo nei Linguaggi Prescrittivi?
2. In cosa si differenziano i registri dei Linguaggi Macchina da quelli del modello della Macchina a Stato.
3. (a) Quali sono i meccanismi del Controllo di Sequenza nei Linguaggi Macchina;
   (b) Come la Translation Semantics mette in corrispondenza il Controllo di Sequenza dei Linguaggi Hight-Level con quelli ei Linguaggi Low-Level
4. Le espressioni in genere conducono a comporre sequenze per contiguità fisica, ovvero l'inizio della successiva è immediatamente dopo la fine della precedente. In quale caso ciò non è vero e perchè?
5. (a) Quali sono i 2 meccanismi da conoscere nell'uso delle espressioni di un Linguaggio di Programmazione?
   (b) Cosa può cambiare in tali meccanismi quando cambiamo Linguaggio?
   (c) Mostrare 1 caso per ogni meccanismo che supporti le risposte date nei punti (a) e (b) precedenti.
6. (a) Si elenchino i 4 tipi di valori visti a lezione
   (b) Si mostri l'uso di ciascuno in 4 frammenti separati di codice C.
   (c) Cosa sono gli l-value ed r-value e a quale dei quattro tipi appartengono
7. (a) Cos'è un comando di iterazione determinata?
   (b) Il linguaggio C possiede un comando di iterazione determinata?
   (c) Cosa possiamo dire se il Linguaggio avesse solo iterazione determinata? Ci dovremmo stupire? Perchè?

Altri esercizi in EserciziL7.pdf