Tipi di Dato III: Strutture Dati Dinamiche

Ivano Salvo – Sapienza Università di Roma

Questa dispensa introduce all'uso di tipi definiti dall'utente, all'importante meccanismo di astrazione fornito dalle *strutture* (o *record*) e soprattutto alla definizione di tipi che definiscono *strutture dati dinamiche*, cioè tipi che i cui elementi vengono continuamente allocati e de-allocati durante l'esecuzione di un programma.

1 Strutture, Enumerazioni e Definizione di Tipi

Una *struttura* è un tipo di dato che permette di memorizzare *dati aggregati*, più universalmente noti in informatica come *record*. Il tipico esempio può essere l'anagrafica di una persona, che contiene varie informazioni di tipo diverso: nome e cognome, data di nascita, luogo di nascita, codice fiscale etc. In tal caso il *tipo* di dato `anagraficaPersona` viene definito come una struttura come segue:

```
struct AP {
    char nome[15];
    char cognome[15];
    data dataNascita;
    char luogoNascita[15];
    char cf[14];
    ...
 };
```

Dopo aver dato questa definizione, `struct AP` è un tipo, per cui posso definire variabili che hanno come tipo `struct AP`: ad esempio posso definire un vettore `struct AP p[100];` dove definisco il vettore `p` di 100 elementi, ciascuno dei quali contiene un record. I sotto–elementi (`nome`, `cognome`, `dataNascita` etc.) vengono detti *campi* e possono venire riferiti via la cosidetta *dot notation*, cioè con il ".", pervasiva nella sintassi dei linguaggi di programmazione. Per esempio, per caricare il nome del 42mo elemento, posso usare l'assegnazione `p[41].nome="paperino"`.

Osserviamo che il tipo `data` del campo `dataNascita` non è un *tipo base* del linguaggio `C` (come `int`, `double`, `char` etc.) ma è un *tipo definito dall'utente*, esso stesso definito a sua volta come una struttura. Una data, come è noto, è composta

da tre interi. Il *millenium bug* e la costante crescita di disponibilità di memoria, ci hanno insegnato che forse non è il caso di economizzare sull'anno (come era tipico in COBOL, dove i numeri venivano memorizzati come sequenze di cifre, e non codificati in binario e quindi era tipico riservare all'anno solo due cifre, come accade nelle date nella forma gg/mm/aa). Per rendere più leggibili i programmi, è opportuno assegnare un *nuovo nome* alla struttura che definisce la tripla di interi. Ciò inoltre permette di "dimenticare" quale sia la struttura del tipo data.

Il C mette a disposizione la possibilità di *definire un nuovo* nome di tipo, attraverso l'istruzione `typedef def-type new-type` dove `def-type` è una definizione di tipo e `new-type` è semplicemente un identificatore, che verrà usato come un tipo che si potrà usare nelle definizioni di variabili e nel passaggio di parametri, senza dover ripetere per esteso la definizione che specifica la forma dei dati di quel tipo. Dal punto vista della chiarezza dei programmi, è chiaramente preferibile definire una variabile con il nuovo tipo `data` che immediatamente richiama il *significato logico* del tipo, piuttosto del più "tecnico" `struct D`.

```
typedef struct D {
            int anno;
            int mese;
            int giorno;
        } data;
```

Un tipo di dato non è semplicemente caratterizzato da un insieme di valori, ma anche da un insieme di operazioni definite su quel dato o che permettono di costruire elementi di quel tipo di dato.

Nel caso delle date, alcune tipiche operazioni potrebbero essere: *inizializzazione* a una certa data di riferimento (per esempio tutti i sistemi di famiglia Unix inizializzano la data del sistema al 1mo gennaio 1970 [questo succede ad esempio dopo uno spegnimento "traumatico"]), *verifica della legalità* di una data (tipicamente quando viene letta da input, è necessario verificare che non venga inserita una data non corretta, come ad esempio 32/12/2012, oppure 29/2/2011), relazioni di *uguaglianza* o di *ordine* tra date (quando definite un nuovo tipo, dovreste sempre, se opportuno, definire operazioni come assegnazione e test di uguaglianza), *differenza* ossia il calcolo del numero di giorni intercorrente tra due date (questa procedura usualmente è di grande utilità in molte applicazioni gestionali), *somma*, ossia determinazione della data fra un certo numero di giorni, e *stampa di un calendario*. Vediamo solo alcuni semplici esempi, giusto per familiarizzare con la sintassi delle strutture, lasciando gli altri esempi alla buona volontà del lettore.

Per rendere più eleganti i nostri programmi, il C mette a disposizione i cosiddetti *tipi enumerati*. Un tipo enumerato permette di definire tipi che sono costituiti da un *numero finito di elementi*. Tipici esempi sono appunto i giorni della settimana, i mesi dell'anno, oppure i sette nani. In realtà, da un punto di vista concreto, in C si

tratta solo di un modo sbrigativo di definire *costanti simboliche*. Una dichiarazione nella forma:

```
enum mesi {gen=1, feb, mar, apr, mag, giu, lug, ago, set, ott, nov, dic};
```

dà la possibilità di definire variabili di tipo `enum mesi`, che avranno come possibili valori `gen`, `feb`, ..., `dic`: in realtà questi valori sono gli interi da 1 a 12 (sarebbero stati gli interi da 0 a 11 se non avessi specificato che si comincia con 1 (`gen=1`)).

È utile ed elegante per scrivere programmi più semplici, definire dei vettori che danno delle informazioni sui singoli mesi (indicizzati con valori da 1 a 12), come il nome o il numero dei giorni (osservate che `m` è un vettore di stringhe):

```
char* m[13]={"", "gennaio", "febbraio", "marzo", "aprile", "maggio",
    "giugno", "luglio", "agosto", "settembre", "ottobre", "novembre",
    "dicembre"};

int dm[13]={0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

È triste dirlo, ma è conveniente definire vettori di questo tipo come *variabili globali*: saranno di fatto trattati come costanti e quindi è un po' eccessivo continuare a passarli come parametri. Date queste definizioni, vediamo tre possibili funzioni di stampa di una data: la prima nella forma tradizionale forma con il nome del mese per esteso, la seconda nella forma `gg/mm/aa`, e l'ultima nella forma `gg/mm/aaaa`.

```
 void printData(data d){
     printf("%2d %s %4d\n",d.g,m[d.m],d.a);
 }

void printDataGGMMAAAA(data d){
    if (d.m<10) printf("%2d/0%1d/%4d\n",d.g,d.m,d.a);
      else printf("%2d/%2d/%4d\n",d.g,d.m,d.a);
}

void printDataGGMMAA(data d){
    if (d.m<10) printf("%2d/0%1d",d.g,d.m);
       else printf("%2d/%2d",d.g,d.m);
    if (d.a%100<10) printf("/0%1d\n",d.a%100);
       else printf("/%2d\n",d.a%100);
}
```

Le uniche cose da osservare sono gli escamotage per stampare degli zeri quando mese e anno hanno 1 sola cifra e l'uso del vettore `m` nella stampa per esteso del mese.

Più interessante vedere una procedura per la lettura di una data dall'esterno, e che controlla che la data immessa sia legale. Da osservare e tenere ben a mente per il prossimo futuro, la notazione `d->g`: si tratta di zucchero sintattico per `(*d).g`: si usa per riferire un campo di una struttura quando abbiamo a disposizione un puntatore a quella struttura. Nel nostro caso abbiamo bisogno di puntatori ai campi della struttura da passare come sempre alla funzione `scanf` e quindi premetteremo l'espressione `d->g` con l'operatore di dereferenziazione `&`:

```
void inserisciData(data *d){
    do {
      printf("\ninserisci il giorno : ");
      scanf("%d",&(d->g));
      printf("\ninserisci il mese : ");
      scanf("%d",&(d->m));
      printf("\ninserisci l'anno : ");
      scanf("%d",&(d->a));
           } while (!legale(*d));
 }
```

La funzione `legale` infine sfrutta pesantemente il vettore `dm` contenente le durate dei mesi. Ricordatevi degli anni bisestili, che sono gli anni divisibili per 4, tranne gli anni divisibili per 100 che sono bisestili solo quando divisibili anche per 400.

```
int bisestile(int anno){
    if (!(anno%4) && (anno%100 || !(anno%400)))
         return 1;
    return 0;
}

int legale(data d){
    int maxg;

    if (d.a<0 || d.m<1 || d.m>12 || d.g<1)
       return 0;
    if (d.m==feb && bisestile(d.a)) maxg=29;
       else maxg=dm[d.m];
    if (d.g>maxg) return 0;
    return 1;
}
```

1.1 Esercizi e Spunti di Riflessione

1. Date la definizione del tipo di dato `razionale` come coppia di interi numeratore/denominatore. Definite la struttura dati e implementate le principali funzioni aritmetiche sui razionali. Rappresentate convenientemente l'output.

2. Date la definizione del tipo di dato `complesso` come coppia di numeri reali. Procedere come nell'esercizio precedente.

3. Un aspetto interessante dei numeri razionali è che lo stesso numero ha diverse possibili rappresentazioni (infinite a dire il vero!). Inoltre, non tutte le coppie di numeri interi sono razionali (ovviamente tutte le coppie $n, 0$ non sono rappresentano nessun numero razionale).

Riflettete sui vantaggi/svantaggi di adottare (e mantenere in memoria) una *rappresentazione canonica* dei numeri razionali.

4. Siccome l'appetito vien mangiando, perchè non definire il tipo di dato `polinomio` (ad esempio come vettore di coefficienti)? E per testare come le astrazioni si compongono, potete lanciarvi sui polinomi a coefficienti razionali, per esempio.

5. Prendete una qualsiasi funzione a vostro piacere, e definite il tipo del suo activation record.

6. Scrivere una funzione che determina se una data è minore di un'altra.

7. Scrivere una funzione che determina il numero di giorni che intercorre tra due date. Se la prima è maggiore della seconda, dare il risultato come numero negativo.

8. Scrivere una funzione che ricevendo in input una data d e un intero n, restituisce in output una data corrispondente alla data che segue d di n giorni.

9. Scrivere una funzione che stampa una data scrivendo anche il giorno della settimana. Ad esempio se la data in ingresso fosse {`2012, 5, 22`}, l'output dovrebbe essere `martedi, 22 maggio 2012`.

10. Scrivere una funzione che prendendo in input un anno e un mese, stampa un "calendario" di quel mese. Ad esempio, prendendo in input 2012 e 5, potrebbe dare in output una stampa in questa forma:

```
maggio 2012
lun mar mer gio ven sab dom
      1   2   3   4   5   6
  7   8   9  10  11  12  13
 14  15  16  17  18  19  20
 21  22  23  24  25  26  27
 28  29  30  31
```

2 Memoria Allocata Dinamicamente e il Tipo `void*`

La memoria che viene allocata a un programma con il frammento del C che conosciamo può essere calcolata *staticamente* (cioè leggendo il codice del programma)[1].

Molti programmi necessitano di una quantità di memoria che dipende dall'esecuzione del programma. Il C permette al programmatore di allocare nuova memoria, grazie ad alcune funzioni di libreria: `malloc` e `calloc`. La funzione `malloc` ha il prototipo `void* malloc(int);`: essa riceve un parametro intero (il numero di byte da allocare) e restituisce come risultato un puntatore alla memoria allocata. Il puntatore tornato, dovendo essere compatibile con qualsiasi tipo puntatore, ha tipo `void*`. Un puntatore di tipo `void*` *è compatibile* per assegnazione con qualsiasi tipo puntatore. Tuttavia, non è possibile referenziare un `void*`: infatti per applicare l'operatore `*` è necessario conoscere il tipo per interpretare correttamente il contenuto della memoria (che è sempre e comunque una sequenza di bit) e inoltre occorre conoscere la *lunghezza* del tipo di dato per sapere quanti bytes andare a leggere (ad esempio, un `char` occupa un byte, un `int` occupa 4 bytes (nei moderni calcolatori, in passato una dimensione più tipica era 2 bytes), una struct occupa (indicativamente) la somma delle lunghezze dei campi, ma non speculate mai su questo fatto). Per dereferenziare una valore di tipo `void*` bisogna prima assegnarlo a una variabile di tipo `T*` con `T` tipo concreto.

La memoria allocata da una chiamata alle funzioni `malloc` e `calloc` si trova in un'area di memoria diversa dallo stack di allocazione delle chiamate di funzione che si chiama *heap*. Va ricordato che qualora non ci fosse più memoria disponibile, una chiamata a `malloc` e `calloc` restituisce il puntatore `NULL`. La prudenza richiederebbe di proteggere ogni chiamata a queste funzioni con un'espressione condizionale che verifica se la memoria è stata effettivamente allocata.

Usate *sempre* l'operatore `sizeof` per determinare la quantità di memoria da allocare. L'operatore `sizeof` è una a pseudo-funzione che accetta come argomento un *tipo* e restituisce il numero di bytes necessario a memorizzare un valore di quel tipo. Ad esempio, se volete allocare dinamicamente memoria per memorizzare una data, scrivete `malloc(sizeof(data))` e non speculate sulla vostra possibilità di calcolare i bytes necessari. Tra l'altro, questo numero potrebbe variare da macchina a macchina oppure tra due versioni diverse dello stesso compilatore. Quindi, usando sistematicamente `sizeof` scriverete programmi che possono essere ricompilati correttamente su diverse architetture e resistenti all'usura del tempo. Anche se dovete allocare memoria per un intero, scrivete *sempre* `malloc(sizeof(int))` e non `malloc(4)`.

Ricordiamo infine la funzione `void * calloc(int n, int d)` che alloca una quantità di memoria sufficiente a contenere n elementi, ciascuno di dimensione d. La memoria allocata è *contigua* (quindi è diverso eseguire n `malloc(d)` consecutive rispetto ad eseguire `calloc(n, d)`). Il fatto che `calloc` allochi un blocco di memoria

[1] questo a essere precisi non è del tutto vero: ad esempio una chiamata ricorsiva alloca memoria per memorizzare gli activation records.

consecutiva ricorda la struttura degli array. Infine, ricordiamo che `calloc` *azzera* il contenuto della memoria allocata.

2.1 Liberazione della Memoria Allocata

La memoria allocata dinamicamente andrebbe *liberata* e resa disponibile per nuove applicazioni ogni qualvolta essa non sia più necessaria. Spesso, l'allocazione dinamica viene tipicamente usata quando un programma continuamente ha bisogno di memoria che poi può venire rilasciata. Un eccellente esempio è lo stack di attivazione delle chiamate di funzione: a ogni chiamata un activation record viene posto sullo stack, e poi rimosso quando la funzione termina la sua esecuzione e lo stack deve essere pronto ad accogliere nuovi activation records. Per restituire all'insieme delle celle libere della memoria allocata (e deve essere stata allocata dinamicamente con `malloc`) esiste la funzione `void free(void*);`.

Se `ptr` è un `T*`, e punta ad un'area di memoria allocata dinamicamente, l'invocazione `free(ptr)` permette di rendere nuovamente disponibili i `sizeof(T)` bytes puntati da `ptr`. Spesso è possibile che assegnando puntatori della memoria rimanga non più *riferibile* (cioè accessibile mediante puntatori): tale memoria è detta *garbage*, cioè spazzatura. Occorre prestare attenzione, e liberare la memoria quando non è più utilizzata, per non saturare la memoria con informazioni non più utilizzabili. Infatti, a differenza di altri linguaggi che producono codice per recuperare periodicamente la memoria non raggiungibile (il cosidetto processo di *garbage collection*, letteralmente "raccolta della spazzatura"), il C pretende (coerentemente nella sua logica di produrre programmi efficienti scritti da programmatori esperti) che il programmatore deallochi esplicitamente la memoria.

L'ultima osservazione è relativa alla funzione `free`: coerentemente con la logica del linguaggio, un'invocazione `free(ptr)` si limita a rendere disponibile per successive `malloc` la memoria puntata da `ptr`, ma non sovrascrive quella memoria. Tuttavia, riferirla *dopo* l'esecuzione di una `free` è da considerarsi un grave errore: in particolare nei moderni sistemi, con molti processi in esecuzione, non possiamo mai escludere che qualche altro processo abbia già riallocato e sovrascritto la memoria che è stata liberata, anche se il nostro programma non esegue nessuna operazione tra la `free` e un ulteriore (azzardato, direi) riferimento a tale memoria.

2.2 Esercizi e Spunti di Riflessione

1. ▶ Scrivere un programma che alloca abbastanza memoria per produrre un errore di overflow della memoria.

2. ▶ Usate l'operatore `sizeof` per scoprire la dimensione di vari tipi di dato, dai tipi di dato predefiniti (`int`, `char`, `int *`, `float`, `long int` etc.) ai tipi di dato definiti da voi (come `data` etc.).

3. ▶ Verificate che dopo aver definito una variabile come `int* a = calloc(n, sizeof(int);` potete usarla esattamente come fosse un vettore, scandendola con l'aritmetica dei puntatori, oppure usando l'operatore `[]`.

4. ▶ Eseguite un po' di `malloc` e verificate se la memoria viene allocata in zone vicine. Verificate anche se i puntatori alla memoria allocata siano "vicini" o "lontani" dai puntatori delle variabili allocate staticamente.

3 Liste

Abbiamo già introdotto il tipo di dato $Seq[A]$ per indicare le sequenze di elementi di un certo tipo A, parlando degli anagrammi. Rivediamo questa definizione:

> Dato un insieme di elementi A, la costante *sequenza vuota* $\langle\,\rangle$ e l'*operazione di concatenazione* $\cdot$, definiamo per induzione le sequenze di elementi di A, Seq[A] come segue:
>
> $$\langle\,\rangle \in \text{Seq}[A]$$
> $$s \in \text{Seq}[A], a \in A \Rightarrow a \cdot s \in \text{Seq}[A]$$
>
> Osservate che l'operazione di concatenazione è una funzione che permette di costruire nuove sequenze a partire da un elemento di A e altre sequenze più corte e quindi è una funzione $\cdot : A \times \text{Seq}[A] \mapsto \text{Seq}[A]$. Data la precedente definizione, una sequenza di elementi $a_1, a_2 \ldots, a_n$ ha la forma $a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot \langle\,\rangle$. Prendiamoci la libertà di indicare una tale sequenza con $\langle a_1, a_2, \ldots, a_n\rangle$, convenendo che $a \cdot \langle a_1, \ldots, a_n\rangle = \langle a, a_1, \ldots, a_n\rangle$.

Proponiamoci ora di implementare questo tipo di dato, che chiameremo d'ora in poi più frequentemente *lista* in C. Considereremo dapprima (e prevalentemente) le liste di interi. Una sequenza non vuota s contiene due elementi: il primo elemento, detto *testa* della sequenza, e una sequenza più corta di sm detta *coda*. Quindi una struttura è una naturale rappresentazione per una sequenza. Un primo tentativo (errato) potrebbe essere il seguente:

```
typedef struct L {
      int testa;
      struct L coda;
   } listanode;
```

La definizione data sopra non è accettabile dal C, perché non è possibile che la definizione di tipo contenga un campo dello stesso tipo che si sta definendo (in effetti questo definirebbe una struttura dati che occupa uno spazio infinito). È necessario aggirare l'intrinseca ricorsività della definizione induttiva di lista, usando l'escamotage dei *puntatori*. Quindi in C non posso definire un campo di tipo `struct L` mentre sto definendo `struct L`, ma posso definire un campo di tipo `struct L*`:

in fondo, un puntatore, altro non è che l'indirizzo di una cella di memoria, indipendentemente da cosa effettivamente punti (il tipo è necessario solo quando vado a leggere la memoria puntata).

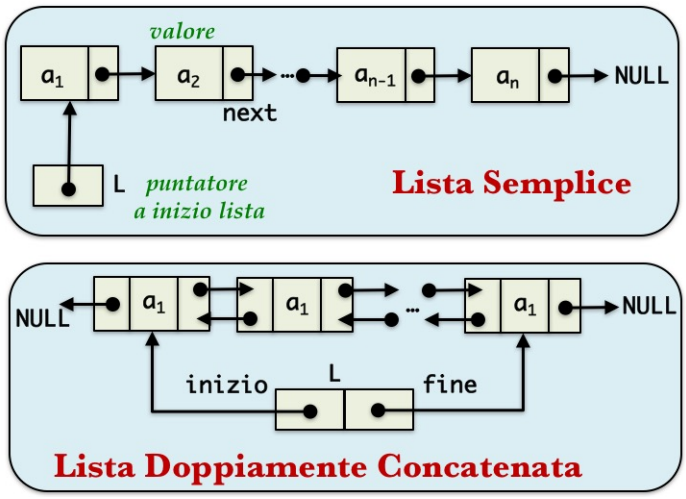

Figura 1: Rappresentazione in memoria di una lista.

L'idea è quella di vedere una lista come una sequenza di *nodi* di lunghezza a priori illimitata[2]. Ciascun nodo della lista contiene un campo per memorizzare l'informazione di tipo A ed un campo per mantenere un puntatore al resto della lista (più concretamente al nodo successivo: per questo campo useremo il nome tradizionale `next`). Il tipo `lista` sarà poi definito come un tipo puntatore ad un nodo: infatti avendo un puntatore all'inizio della lista ed inseguendo i puntatori posso accedere (sequenzialmente) a tutti gli elementi della lista. Coerentemente la lista vuota sarà rappresentata dalla costante pointer `NULL`.

```
typedef struct L{
     int val;
     struct L * next;
} listanode;

typedef listanode* lista.
```

Ovviamente la rappresentazione scelta, anche se è la più comune in letteratura, non è la sola possibile. Nel seguito vedremo anche un'altra rappresentazione per implementare in modo efficiente alcune operazioni (vedi Fig. 1). Nella trattazione che segue, considereremo sempre due aspetti:

[2] l'unica limitazione sarà data dalla disponibilità di memoria da allocare ad eventuali nuovi nodi.

```
lista cons(lista L, int x){
  lista lAux = (lista)malloc(sizeof(listanode));
      /* alloco memoria per un nuovo nodo */
  lAux->val = x;    /* lAux = <el> */
  lAux->next = L;   /* lAux = el.L */
  return lAux;
}
```

Figura 2: Costruttore C `cons` del tipo `lista`.

1. un aspetto *logico*, che a che fare con cosa sia una lista, quali siano i valori di tipo lista (la lista vuota, distinta dalle liste non vuote, formate da un elemento (*testa*) seguito dal resto della lista (*coda*));
2. un aspetto *implementativo*, legato ad una *rappresentazione* nel linguaggio di programmazione scelto[3], nel nostro caso il linguaggio C.

Cercheremo di scrivere programmi che siano il più aderenti possibile all'aspetto logico, seguendo la seguente strategia: scriveremo dapprima una serie di funzioni sulle liste che implementano le operazioni elementari (come aggiungere o togliere un elemento) per poi dimenticare, per quanto possibile, le operazioni troppo a basso livello (come allocare o deallocare memoria o spostare un puntatore) che non appartengono alla specifica logica del tipo di dato, ma alla sua rappresentazione: nella scrittura di funzioni più complesse potremmo riferirci alle funzioni definite come *azioni elementari*, scrivendo programmi più legati all'aspetto logico che a quello implementativo e quindi più vicino alle specifiche e al nostro modo di concettualizzare i problemi, cercando quindi di scrivere programmi sperabilmente più leggibili.

Occasionalmente, può risultare utile considerare l'aspetto implementativo per questioni di efficienza. Spesso, infatti, l'accesso diretto alle componenti di un tipo di dato può permettere implementazioni più efficienti, purtroppo legate all'implementazione concreta e non alla logica di un tipo di dato.

3.1 Tipo Lista Semplice: *Costruttori* e *Distruttori*

Come evidenziato dalla definizione induttiva di sequenza, per *costruire* una qualsiasi lista, è sufficiente la costante lista vuota $\langle\rangle$ (rappresentata in C semplicemente dal puntatore `NULL`) e l'operazione $\cdot$ che permette di aggiungere un elemento in testa alla sequenza: funzioni di questo tipo che permettono di generare un qualsiasi elemento di un tipo di dato vengono tradizionalmente chiamate, nella Teoria dei Tipi, *costrut-*

[3] osserviamo a questo proposito che esistono linguaggi di programmazione in cui i tipi di dato vengono specificati semplicemente attraverso la loro struttura logica, lasciando al compilatore il fardello di costruire una adeguata rappresentazione interna *inaccessibile* al programmatore.

```
int isEmpty(lista L){
   /* POST: torna 1 se L = <>
    * 0 altrimenti
    */
  if (L==NULL) return 1
  else return 0;
}
```

```
int head(lista L){
   /* PREC: L != <> */
  return L->val;
}
```

```
int tail(lista L){
   /* PREC: L != <> */
  return L->next;
}
```

Figura 3: Distruttori C del tipo `lista`.

tori[4]. Osserviamo che, ad esempio, per i numeri naturali i costruttori sarebbero la costante 0 e il successore. Chiameremo (per motivi storici legati al linguaggio LISP) `cons` (*constructor*, appunto) la funzione C che permette di creare una nuova lista aggiungendo un elemento in testa (in Fig. 2 l'implementazione, quindi, del costruttore $\cdot$ nella specifica logica delle sequenze).

Viceversa, per definire funzioni per *ricorsione* sulla struttura induttiva delle liste (o più in generale di ogni tipo di dato induttivo), è necessario:

1. distinguere le possibili forme di una lista (lista vuota e lista non vuota);
2. accedere alle componenti di una lista (elemento in testa, e coda della lista).

Anche qui, a ben riflettere, c'è una chiara analogia con le funzioni ricorsive sui naturali, in cui in generale è necessario distinguere se un naturale è 0 (caso base) oppure un successore $n+1$ (caso induttivo), e la definizione ricorsiva dipende in questo secondo caso dal numero n. *Quasi* (!?) tutte le funzioni "interessanti" sui naturali $f : \mathbb{N} \to A$ possono essere infatti definite per *ricorsione primitiva*, cioè definendo una funzione f come soluzione di equazioni ricorsive nella seguente forma:

$$\begin{aligned} f(0) &= c \\ f(n+1) &= g(f(n), n) \end{aligned}$$

dove $c \in A$ e $g : A \times \mathbb{N} \to A$.

Le funzioni che *decompongono* un tipo di dato complesso, si chiamano tradizionalmente *distruttori*. Il codice C che definisce i distruttori del tipo lista è mostrato in Fig. 3. A titolo di ripasso del linguaggio C è interessante vedere due possibili codici alternativi per la semplice funzione `isEmpty` in Fig. 5

Un'interessante alternativa, più nelle corde della programmazione C, sarebbe scrivere un *unico* distruttore, che contemporaneamente verifica se una lista sia vuota e in caso negativo, estrae le sottocomponenti di una lista (testa e coda).

[4] per la precisione, ogni lista, può essere scritta in modo *unico* come una sequenza di applicazioni dei costruttori $\langle\rangle$ e $\cdot$

```c
int isEmpty(lista L){
  if (L) return 1;
  return 0;
}
```

```c
int isEmpty(lista L){
  return !L;
}
```

Figura 4: Codici da Vero Programmatore C per `isEmpty`.

```c
int isEmpty(lista L, int* h, lista* T){
   if (!L) return 0;
   *h = L->val;
   *T = L->next;
   return 1;
}
```

Figura 5: Distruttore *unico* per le liste

Osservate che il parametro `T` essendo di tipo `lista*`, di fatto è un `listanodo**`. Usualmente abbiamo usato i passaggi per riferimento per produrre *side-effects* sul chiamante. Dovendo modificare un dato intero passavamo un puntatore ad un intero. Ma questa volta, *vogliamo modificare un dato puntatore*, e quindi è necessario passare *un puntatore a un puntatore*.

Ritornando all'aspetto logico, le semplici funzioni `cons`, `isEmpty`, `head` e `tail` ci permettono di scrivere tutte le funzioni calcolabili sulle liste senza scrivere nessun puntatore e nessun operatore su di essi! Vediamo alcuni semplici esempi, prima *specificando* il comportamento delle funzioni con *equazioni ricorsive* e poi traducendo queste in C, in analogia con quanto fatto con le funzioni ricorsive sui naturali. Cominciamo con le funzioni *length* (lunghezza di una lista), *sumL* (somma degli elementi della lista), *maxL* (massimo valore memorizzato in una lista), e *twiceL* (che calcola una lista i contenente gli elementi della lista originaria moltiplicati per 2).

$$\begin{aligned} \mathsf{length}(\langle\,\rangle) &= 0 \\ \mathsf{length}(h \cdot t) &= 1 + \mathsf{length}(t) \end{aligned} \qquad \begin{aligned} \mathsf{sumL}(\langle\,\rangle) &= 0 \\ \mathsf{sumL}(h \cdot t) &= h + \mathsf{sumL}(t) \end{aligned}$$

$$\begin{aligned} \mathsf{maxL}(\langle\,\rangle) &= -\infty \\ \mathsf{maxL}(h \cdot t) &= \mathsf{max}(h, \mathsf{maxL}(t)) \end{aligned} \qquad \begin{aligned} \mathsf{twiceL}(\langle\,\rangle) &= \langle\,\rangle \\ \mathsf{twiceL}(h \cdot t) &= 2h \cdot \mathsf{twiceL}(t) \end{aligned}$$

Le traduzioni in C delle funzioni viste sopra sono essenzialmente una questione di sintassi. Una possibile versione in Fig. 6. Come sempre, i diversi casi nella definizione induttiva di una funzione[5] vengono semplicemente tradotti con un costrutto

[5]qui, ad essere precisi si tratta di *induzione strutturale* sulla struttura della lista, ma potete sempre pensare che ci sia una qualche misura sui numeri naturali che decresce nella parte destra delle equazioni ricorsive, e in questo caso è semplicemente la lunghezza della lista.

```
int length(lista L){
   if (isEmpty(L)) return 0;
   return 1+length(tail(L));
 }
```

```
int sumL(lista L){
   if (isEmpty(L)) return 0;
   return head(L)+sum(tail(L));
 }
```

```
int maxL(lista L){
   int m;
   if (isEmpty(L)) return MININT;
   m = maxL(tail(L));
   if(head(L)>m) return head(L);
   return m;
}
```

Figura 6: Semplici funzioni sulle liste in C (versione funzionale).

condizionale che distingue le liste vuote da quelle non vuote.

Probabilmente, un Vero Programmatore C si sente più a suo agio con le versioni in Fig. fig:simplyListVPC, dove si evita di chiamare una funzione per verificare se una lista è vuota e per estrarre testa e coda della lista. Ovviamente, tali funzioni, speculano sul fatto di *conoscere l'implementazione* delle liste e quindi sono scritte nei termini della rappresentazione *concreta* a puntatori delle liste in C piuttosto che ragionando sulle operazioni rese disponibili sul tipo di dato lista.

Da un punto di vista concreto, in questi semplici esempi, non fa molta differenza, ma, a volte, poter accedere ai dettagli implementativi potrebbe essere nocivo, in quanto permette a dei programmatori utenti di distruggere delle assunzioni (dette *invarianti di tipo di dato*) fatte dai programmatori di una libreria.

3.2 Trasparenza Referenziale

Abbiamo volutamente dimenticato l'implementazione C della funzione *twiceL*, che merita un discorso a parte. La funzione *twiceL* ci mostra che, quando dobbiamo

```
int length(lista L){
   if (!L) return 0;
   return 1+length(L->next);
 }
```

```
int sumL(lista L){
   if (!L) return 0;
   return L->val+sum(L->next);
 }
```

```
int maxL(lista L){
   int m;
   if (!L) return MININT;
   m = maxL(L->next);
   if(L->val>m) return L->val;
   return m;
}
```

Figura 7: Semplici funzioni sulle liste in C (versione VPC).

```
lista twiceL(lista L){
  if (isEmpty(L)) return emptyList;
  return cons(2*head(L),
              twiceLFun(tail(L)));
    }
```

```
void twiceLRec(lista L){
   if (L) {
      L->val = 2 * L->val;
      twiceLRec(L->next);
      }
}
```

Figura 8: Funzione *twiceL* in C: versione che genera una nuova lista e in place.

costruire una lista come risultato, $\cdot$ e $\langle\ \rangle$ sono tutto quello di cui abbiamo bisogno. Parallelamente, nelle nostre traduzioni in C ci basteranno la costante `NULL` e `cons`.

Tuttavia, tra la funzione nel mondo incontaminato e platonico della "semantica" e la sua implementazione in C si aprono diversi scenari. Una funzione `twiceL` in C infatti, può comportarsi in due modi molto diversi. Potremo infatti scrivere sia una funzione `twiceL` che *genera* una nuova lista, lasciando la lista originale immutata, oppure una funzione `twiceL` che *modifica* la lista originaria. Non c'è una soluzione corretta. Dipenderà dal problema che stiamo risolvendo, o, se preferite, dalle *specifiche*.

Traducendo le equazioni ricorsive usando `cons` per tradurre $\cdot$ otterremo la funzione che genera una nuova lista, perché `cons` *alloca nuova memoria*. La funzione `twiceLFun` è mostrata in Fig. 8. Sempre in Fig. 8 è mostrata anche la funzione ricorsiva `twiceLRec` che modifica la lista in ingresso. La abbiamo definita di tipo `void`, in quanto il pointer al primo elemento della lista rimane immutato. Altre volte sarà conveniente, anche scrivendo funzioni che lavorano per side-effects, definirle di tipo `lista`, quando il puntatore di inizio lista può potenzialmente cambiare (ciò accade sempre quando si inseriscono o rimuovono elementi).

Ovviamente, nessuno ci vieta di scrivere queste funzioni iterativamente, come mostrato in Fig. 9, anche se lo stile ricorsivo meglio si presta, *usualmente* a scrivere programmi su strutture dati induttive (o ricorsive).

```
lista twiceLFunIt(lista L){
  lista lPtr=L;
  lista res=emptyList;
  while (lPtr){
      res = cons(2*head(lPtr), res);
      lPtr = lPtr->next;
      }
  return res;
  }
```

```
void twiceLIt(lista L){

   lista lPtr=L;
   while (lPtr){
    lPtr->val *= 2;
    lPtr = lPtr->next;
    }
 }
```

Figura 9: Funzione *twiceL* in C (versioni iterative).

3.3 Inserimento di Elementi

Cominciamo con il vedere una funzione che aggiunge un elemento in coda ad una lista. Questa funzione si può facilmente scrivere osservando che aggiungere un elemento in testa o in coda alla lista vuota produce lo stesso effetto. Quindi ci aspettiamo che siano verificate le seguenti equazioni ricorsive:

$$\begin{aligned} \mathsf{addTail}(\langle\,\rangle, x) &= \langle\, x \,\rangle \; [= x \cdot \langle\,\rangle\,] \\ \mathsf{addTail}(h \cdot t, x) &= h \cdot \mathsf{addTail}(t, x) \end{aligned}$$

da cui si deriva facilmente il seguente codice C:

```
lista addTailFun(lista L, int x){
   if (isEmpty(L)) return cons(L, x);
   return cons(addTailFun(tail(L), x), head(L));
}
```

Probabilmente il lettore non è familiare con questo stile di programmazione (e non lo sarebbe neanche il nostro amico immaginario, il Vero Programmatore C), fatto solo di composizione di chiamate di funzione, eventualmente annidate, tipico dei linguaggi detti appunto *funzionali*.

C'è inoltre un altro aspetto da considerare: la funzione `cons` *alloca nuova memoria*, quindi la funzione `addTailFun` in realtà *crea* una *nuova* lista, lasciando inalterata la lista di ingresso `L`. Questo, potrebbe essere talvolta desiderabile, mentre altre volte il comportamento atteso è semplicemente la *modifica* della vecchia lista.

Tenete presente che, chiamando la funzione `addTailFun` con una istruzione del tipo `L=addTailFun(L, x);` la vecchia lista `L` viene irrimediabilmente perduta (a meno che non ci siano altre variabili nel programma *alias* di `L`) ed la memoria inizialmente puntata da `L` diventa *garbage*, cioè spazzatura, ossia memoria allocata al vostro programma, ma che il programma non può più riferire.

Questa continua allocazione di nuova memoria è un *comportamento voluto* nei linguaggi funzionali, perchè garantisce la cosiddetta *trasparenza referenziale*, cioè come le variabili della matematica, all'interno di un'espressione variabili uguali riferiscono cose uguali, mentre come abbiamo già visto questo non è vero per le variabili C, in quanto la valutazione di una sotto-espressione (semplicemente un `x++` per esempio, senza scomodare la chiamata di funzione) *può modificare* il valore di alcune variabili, facendo sì che durante la valutazione di una stessa espressione la stessa variabile possa valutare a valori diversi. Tuttavia, durante l'esecuzione dei programmi scritti nei linguaggi funzionali, c'è costantemente attivo un processo di *garbage collection* che rende nuovamente disponibile la memoria che non è più riferibile.

In un linguaggio imperativo, disponendo dell'assegnazione, un codice più ragionevole potrebbe essere il seguente (tenete presente anche che un Vero Programmatore C non invocherebbe mai una funzione per un compito così stupido come verificare se

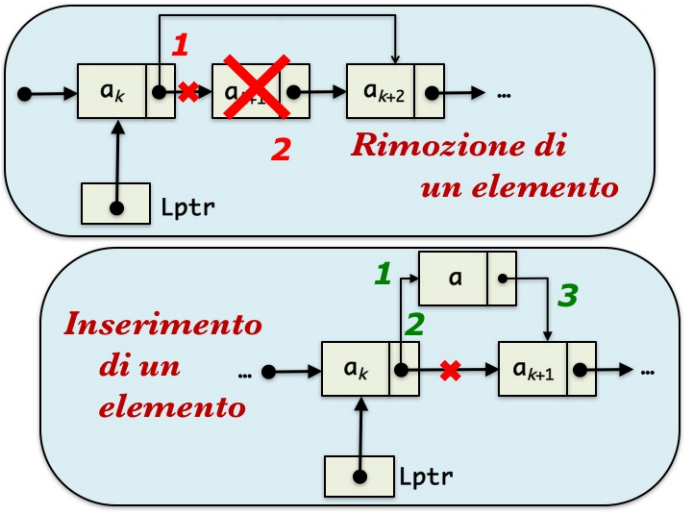

Figura 10: Inserimento e rimozione in una lista (versione a *side-effects*).

un puntatore sia `NULL` o meno. Neanche sotto tortura!). La funzione `addTailRec`, a differenza di `addTailFun` *alloca un solo nuovo nodo*.

```
lista addTailRec(lista L, int x){
   if (!L) return cons(L, x);
   L->next = addTailRec(L->next, x);
   return L;
}
```

Nel seguito, per facilitare la lettura, useremo sempre la convenzione di dare un nome con il suffisso `Fun` alle funzioni che, dovendo restituire una lista come risultato, *allocano una nuova lista*, e il suffisso `Rec` per le funzioni ricorsive che *producono side-effects* su liste esistenti. Le (poche) versioni iterative che vedremo avranno nel nome il suffisso `It`, e in genere si comporteranno producendo side effects su liste esistenti (anche se ovviamente, anche le funzioni iterative potrebbero appartenere alla famiglia `Fun`, creando nuove liste, come abbiamo visto nell'esempio di `twiceLItFun`).

Per concludere questa breve esplorazione dei possibili trattamenti di una lista, vediamo infine un'ultima versione dell'inserimento in coda che ci ricorda che tutto quello che si può fare ricorsivamente, si può anche fare iterativamente. Anche se in questo caso inseriamo il nuovo elemento in coda, già si intuisce che per inserire un elemento iterativamente è *necessario* avere a disposizione *due puntatori*: uno al nodo che precederà il nuovo elemento inserito, e uno al nodo seguente (che in questo caso è `NULL`). In realtà, avendo a disposizione il precedente posso accedere al seguente attraverso il pointer `next`, basterà posizionarsi sul precedente (vedi Fig. 10). Nel caso ricorsivo, possiamo sempre speculare sul fatto che il nodo precedente sia memorizzato nell'activation record della funzione che ha generato la chiamata, e che verrà

ripreso in considerazione *al rientro* (infatti, ad esempio, nella funzione `addTailRec` il corretto *aggancio* viene fatto dall'istruzione `L->next=addTailRec(L->next, x);` che viene eseguita *al rientro* della chiamata ricorsiva).

Nell'inserimento in coda iterativo, è necessario scorrere la lista (senza dimenticare il puntatore di inizio lista) per posizionarsi in fondo, fermarsi all'ultimo nodo (prima di arrivare a `NULL`) e finalmente attaccare il nuovo nodo creato. Personalmente (e sono in buona compagnia) ne deduco che il modo corretto di scrivere programmi sulle liste è la ricorsione, a meno di casi *molto* particolari.

```
lista addTailIt(lista L, int x){
   lista Laux = L;
   lista temp = cons(NULL, x);
      /* creo il nuovo nodo da inserire */

   if (Laux) {
         while (Laux->next) Laux=Laux->next;
            /* mi posiziono sull'ultimo nodo */
         Laux->next=temp;
            /* aggancio il nuovo nodo */
         return L;
   } else return temp;
        /* cambia il pointer di inizio lista */
}
```

3.4 Concatenazione di Liste

Per corroborare le importanti questioni affrontate nella sezione precedente, proseguiamo i nostri primi passi sulle liste, con un altro tipico problema: date due liste, appendere la seconda lista alla prima. Di fatto questo problema è del tutto analogo all'inserimento in coda.

Ancora una volta, possiamo specificare mediante equazioni ricorsive le relazioni logiche verificate dalla funzione append (è sufficiente fare induzione sul primo parametro):

$$\begin{aligned} \mathsf{append}(\langle\,\rangle, s) &= s \\ \mathsf{append}(h \cdot t, s) &= h \cdot \mathsf{append}(t, s) \end{aligned}$$

da cui si ricava immediatamente il corrispondente codice C in stile "funzionale":

```
lista append(lista L, lista M){
   if (isEmpty(L)) return M;
      else return cons(head(L), append(tail(L), M));
}
```

Per quanto riguarda la correttezza di questa funzione, è banale verificare che esse calcolano quanto prescritto dalle specifiche mediante equazioni ricorsive. La terminazione invece dipende solo dal fatto che le chiamate ricorsive vengono attivate con parametri che contengono liste "più corte": una ovvia funzione di terminazione per `addTail` o `append` sarà la lunghezza della lista passata come primo parametro.

Osservate che `append`, come `addTail`, crea nuova memoria decomponendo e ricostruendo la lista `L`, ma *non alloca nuova memoria* per copiare `M`. Quindi non merita il suffisso `Fun` nel nome, avendo un comportamento "misto", direi in generale da evitare. Per ottenere un "vero" comportamento funzionale, dovremmo scrivere:

```
lista appendFun(lista L, lista M){
   if (isEmpty(L)) return copy(M);
      else return cons(head(L), appendFun(tail(L), M)));
}
```

dove `copy` è la funzione definita dal seguente codice:

```
lista copy(lista L){
   if (isEmpty(L)) return L;
      else return cons(head(L), copy(tail(L)));
}
```

Da un punto di vista *logico*, `copy` infatti è semplicemente la funzione *identità* sulle liste. Ovviamente, mentre nel mondo incontaminato della semantica dopo $L = \mathsf{copy}(M)$, L ed M sono semplicemente due identificatori per lo stesso valore, computazionalmente fa molta differenza avere due zone della memoria, che *incidentalmente* memorizzano lo stesso valore.

Come prima, possiamo scrivere funzioni che si limitano a *spostare un puntatore* e quindi a legare l'ultimo elemento di `L` con la testa di `M`. Chiameremo in generale queste funzioni *in place*.

```
lista appendRec(lista L, lista M){
   if (!L) return M;
   L->next = appendRec(L->next, M);
   return L;
}
```

Dovete essere però pienamente consapevoli del fatto che dopo un'operazione tipo `appendRec` eventuali modifiche su `M` si ripercuoteranno sulla nuova lista! Chiariamo la cosa con un esempio: supponete che `L` memorizzi la sequenza $\langle 1, 2, 3\rangle$ ed `M` memorizzi la sequenza $\langle 4, 5, 6\rangle$. Consideriamo i seguenti frammenti di codice:

```
L = appendFun(L,M);
M = addTRec(M,7);
```

```
L = appendRec(L,M);
M = addTRec(M,7);
```

Dopo l'esecuzione di questi due comandi, nel caso a sinistra `L` memorizzerà la lista $\langle 1,2,3,4,5,6\rangle$, mentre a destra mentre nel caso a destra memorizzerà la lista $\langle 1,2,3,4,5,6,7\rangle$ (vedi figura 11). Cosa sia giusto e cosa sia sbagliato, dipenderà dai vostri obiettivi!

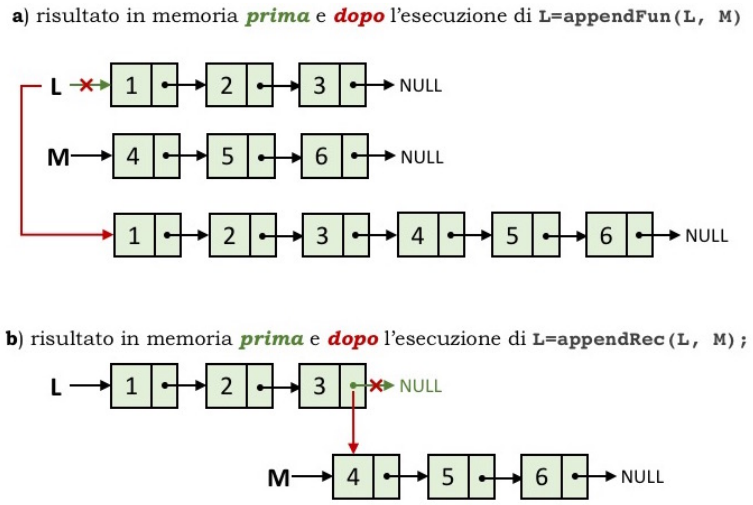

Figura 11: Effetto di append funzionale e per side-effects.

La versione iterativa di questo problema è del tutto analoga alla versione iterativa dell'aggiunta in coda e viene pertanto lasciata come esercizio.

3.5 Rovesciamento di una Lista

Proponiamoci ora di scrivere una funzione che rovescia gli elementi di una lista. È relativamente semplice scrivere la specifica logica, attraverso equazioni ricorsive. Chiaramente, la lista vuota rovesciata è ancora la lista vuota, mentre la testa della lista va appesa in coda al rovesciamento della coda della lista (i primi saranno gli ultimi, dopo il rovesciamento):

$$\begin{aligned} \mathsf{reverse}(\langle\,\rangle) &= \langle\,\rangle \\ \mathsf{reverse}(a \cdot s) &= \mathsf{addTail}(\mathsf{reverse}(s), a) \end{aligned}$$

Ancora una volta, le equazioni ricorsive si trasformano facilmente in codice C:

```
lista reverseFun(lista L){
   if (isEmpty(L)) return NULL;
   return addTailRec(reverseFun(tail(L)), head(L));
   }
```

Osserviamo due cose:

1. la funzione, come le sue sorelle marcate `Fun`, alloca nuova memoria per creare una nuova lista (per effetto delle chiamate alla funzione `addTailRec`), mantenendo immutata la lista di ingresso (cosa accadrebbe viceversa se chiamassi la funzione `addTailFun`?);
2. la funzione scorrerà tutta la lista di ingresso, eseguendo quindi un numero di chiamate ricorsive pari al numero di elementi della lista di ingresso.

Tuttavia osserviamo che la chiamata ad `addTailRec` non ha complessità costante, ma proporzionale alla lunghezza della lista a cui attaccare in coda l'elemento; quindi la complessità della funzione, essendo n la lunghezza della lista, sarà $1+2+...+(n-1) = \frac{n \times (n-1)}{2}$, cioè $\mathcal{O}(n^2)$. È questa la complessità intrinseca del problema o è possibile fare meglio?

Intuitavamente, per rovesciare una pila di fogli sul tavolo, basta trasferirli in un unica passata su un'altra pila, facendo n operazioni. Se riuscissimo a costruire la nuova lista facendo n inserimenti in testa, invece che in coda, otterremo una funzione di complessità *lineare*, cioè proporzionale ad n. Per non dimenticare l'iterazione, vediamo l'algoritmo iterativo lineare (anche questo algoritmo alloca una nuova lista):

```
lista reverseItFun(lista L){
   lista M = NULL;
      /* variabile ausiliaria per scorrere L */
   lista Laux = L;

   while (LAux){
     /* INV: L = LAux@reverse(M)) */
      M = cons(M, N->val);
      Laux = Laux->next;
   }
   return M;
}
```

Osservate l'invariante e osservate che all'uscita del ciclo (che avviene quando il pointer `N` assume il valore `NULL`, cioè quando $N = \langle\rangle$), ho che l'asserzione finale è verificata (infatti ho che: $\mathsf{reverse}(L) = M$ se e solo se $L = \mathsf{reverse}(M)$).

Ricordando l'esempio di fibonacci, non è difficile pensare ad un programma ricorsivo che emula il comportamento del programma iterativo, memorizzando su un

parametro ausiliario la lista `M` che crea il programma iterativo, potendo così *trasmettere alle future chiamate ricorsive* i valori via via calcolati. Per rispettare l'interfaccia della funzione `reverse`, consideriamo una funzione ausiliaria con due parametri.

```
lista reverseEffAux(lista L, lista M){
   if (isEmpty(L)) return M;
   return reverseEffAux(tail(L), cons(M, head(L)));
}

lista reverseEff(lista L){
   return reverseEffAux(L, NULL);
}
```

Come già visto con i vettori, la funzione `reverseEffAux` usa un trucco che si può rivelare spesso utile: l'osservazione cruciale è che scorrendo una lista ricorsivamente, la si percorre di fatto due volte: una all'"andata" della scansione ricorsiva e una al "ritorno" (in ordine rovesciato!), al rientro delle chiamate ricorsive. Per convincersi "sperimentalmente" di questo fatto (già osservato approposito dei vettori, ma *repetita juvant*), il lettore è invitato a verificare l'output del seguente programma:

```
void printList(lista L){
    if (!L) printf("\n");
      else {
        printf("%d", L->val);
        printList(L->next);
        printf("%3d", L->val);
      }
}
```

È necessario il parametro ausiliario nella funzione `reverseEffAux`, perché ovviamente disponiamo dei risultati calcolati ricorsivamente solo durante il rientro delle chiamate ricorsive. Gli inserimenti in testa alla lista ausiliaria `M` avvengono viceversa all'andata nello scorrimento della lista.

Concludiamo il discorso sul problema del rovesciamento di una lista, ponendoci il problema di effettuare il rovesciamento della lista in place, cioè *senza allocare nuova memoria*. È necessario rovesciare i puntatori. Chiaramente in questo caso distruggeremo la lista originaria. Sfruttando sempre l'idea dell'"andata" e "ritorno", la versione ricorsiva, è leggermente più complicata e richiede un po' di "luccicanza".

La funzione `reverseRec` si limita all'"andata" a trovare il nuovo primo elemento della lista (che è l'ultimo della lista in ingresso) e al "ritorno" sposta opportunamente i puntatori. Forse la Fig. 12 può aiutare a "vedere" cosa accade: sono mostrati l'arrivo all'ultimo elemento (a) e il rientro dalle ultime due chiamate ricorsive (b)-(c). La parte verde allude ai dati modificati nell'attivazione presa in considerazione.

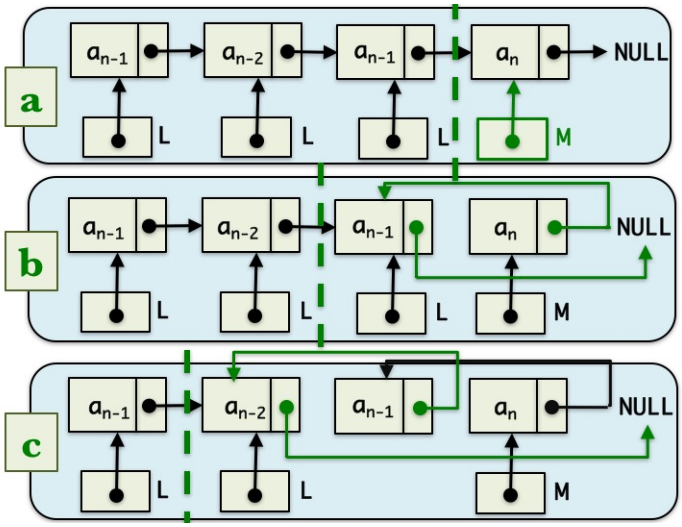

Figura 12: Rovesciamento in place di una lista.

```
lista reverseRec(lista L){
   lista M;

    /* rovescia la lista vuota o con 1 elemento */
   if (!L || !L->next) return L;
    /* se la lista ha almeno 2 elementi */
   M = reverseRec(L->next);
   (L->next)->next = L;
   L->next=NULL;
   return M;
}
```

Osserviamo che i casi base della ricorsione, per una volta, sono 2 (lista vuota e lista che contiene 1 solo elemento) e non semplicemente la lista vuota, sulle quali il rovesciamento non produce effetti. La procedura `reverseRec` tratta correttamente il caso di lista vuota, invoca la funzione ausiliaria che rovescia i puntatori di tutti gli elementi, tranne quello del primo nodo, che essendo diventato l'ultimo, dovrà diventare il pointer `NULL`, il consueto segnale di fine lista.

3.6 Rimozione di Elementi

Vediamo in questa sottosezione, alcuni programmi che rimuovono elementi da una lista. Cominciamo con il programma che rimuove tutte le occorrenze di un certo valore da una lista. Come sempre cominciamo dalla *specifica funzionale* mediante

equazioni ricorsive:

$$\begin{aligned} \mathsf{remove}(\langle\,\rangle, x) &= \langle\,\rangle \\ \mathsf{remove}(a \cdot s, x) &= \begin{cases} a \cdot \mathsf{remove}(s, x) & \text{se } a \neq x \\ \mathsf{remove}(s, x) & \text{se } a = x \end{cases} \end{aligned}$$

Al solito, sintetico e semplice il programma C ricorsivo corrispondente:

```
lista removeFun(lista L, int x){
   if (isEmpty(L)) return NULL;
   if (head(L)==x) return removeFun(tail(L), x);
   return cons(removeFun(tail(L), x));
}
```

Al solito, questo programma alloca una *nuova lista* in cui sono state rimosse le occorrenze del valore x. Più tipicamente, si vorrà che una funzione di rimozione operi sulla lista, *liberando* la memoria allocata ai nodi della lista che vengono rimossi. Sarà sufficiente scorrere la lista e, quando il nodo contiene il valore fornito in input, deallocare la memoria avendo cura di risistemare opportunamente i puntatori.

La deallocazione della memoria avviene attraverso la funzione di libreria `free` che restituisce all'insieme della memoria libera la memoria puntata da un certo puntatore. Osservate che prima di invocare la funzione `free` è necessario salvare *il contenuto* della memoria deallocata, nel caso in cui questo sia utile (tipicamente questo è sempre necessario per sistemare opportunamente i puntatori dopo la cancellazione). La deallocazione in genere non implica la *cancellazione* del contenuto delle celle liberate, ma *non* è prudente speculare sul fatto che queste celle non siano già state riallocate da altre procedure e quindi già sovrascritte con altri dati (ricordate che nei calcolatori moderni, tipicamente molti programmi sono contemporaneamente in esecuzione). Vediamo ancora una volta sia la versione ricorsiva (qui sotto) che iterativa (in Fig. 13).

```
lista removeRec(lista L, int x){
   lista M;

   if (!L) return NULL;
   if (L->val==x){
       M = L->next;
       free(L);
       return(removeRec(M, x));
   }
   L->next = removeRec(L->next, x);
   return L;
 }
```

```
lista removeIt(lista L, int x){
   lista inizioL = L;
   lista currL, precL, tmp;

      /* rimuove tutte le occorrenze iniziali di x. */
   while (inizioL && inizioL->val == x){
      tmp = inizioL->next;
      free(inizioL);
      inizioL = tmp;
    }
    precL = inizioL;
      /* alla prima iterazione precL->val != x */
    while (precL){
       currL = precL->next;
       if (currL && currL->val==x){
             precL->next = currL->next;
             free(currL);
          }
        precL = currL;
    }
    return inizioL;
 }
```

Figura 13: Rimozione Iterativa di un Elemento.

La versione iterativa risulta molto complicata, in quanto bisogna preventivamente determinare l'inizio della lista risultato (eliminando una eventuale sequenza iniziale dell'elemento da eliminare). Inoltre, per rimuovere correttamente un elemento, è necessario avere *due puntatori* a due elementi consecutivi della lista. Per comprendere questo concetto, magari ridate uno sguardo alla Fig. 10.

Osservate ancora una volta che nella funzione `removeRec` la determinazione dell'inizio della nuova lista è in qualche senso *automatica* in quanto ogni attivazione ricorsiva di `removeRec` fornisce come output un corretto nuovo inizio lista, l'attivazione chiamante va correttamente a riagganciare al resto della lista (istruzione `L->next = removeRec(L->next, x);`).

Differenza di Liste. La funzione che rimuove tutte le occorrenze di un certo valore può essere usata convenientemente per scrivere con relativa facilità altre funzioni che rimuovono elementi da una lista. Qui vedremo la funzione diff che prese due sequenze s_1 ed s_2, restituisce una sequenza composta da tutti gli elementi che appartengono ad s_1 ma non a s_2. Come sempre cominciamo dalla specifica funzionale, continuamo con la sua implementazione funzionale `diffFun` e infine vediamo la versione `diffPlace` che opera direttamente sulle liste avute in input modificandole.

$$\begin{aligned} \mathsf{diff}(s_1, \langle\,\rangle) &= s_1 \\ \mathsf{diff}(s_1, a \cdot s_2') &= \mathsf{diff}(\mathsf{remove}(s_1, a), s_2') \end{aligned}$$

In questo caso, curiosamente, le funzioni `diffFun` e `diffPlace` si ottengono entrambe nello stesso modo dalle equazioni ricorsive scritte sopra, semplicemente chiamando le due diverse versioni di `remove`. Il motivo è che l'allocazione di nuova memoria dipende dalla traduzione dell'operatore $\cdot$ sulle sequenze con la funzione `cons` (e di conseguenza dall'uso di `removeRec` o `removeFun`).

```
lista diffRec(lista L, lista M){
   if (!M) return L;
   return diffRec(removeRec(L, M->val), M->next);
}
```

La versione che alloca nuove liste può essere ottenuta semplicemente sostituendo la chiamata di `removeRec` con la chiamata a `removeFun`, che effettivamente alloca una nuova lista, lasciando `L` incontaminata. Purtroppo, in realtà il semplice utilizzo di `removeFun` alloca molta più memoria del necessario. Scriviamo una versione più efficiente, basata su un altro schema ricorsivo, che ci permette anche di continuare a familiarizzare con i concetti introdotti, scriviamo una funzione `diffFun`. In questo caso, faremo ricorsione sul *primo* parametro.

$$\begin{aligned} \mathsf{diff}_2(\langle\,\rangle, s_2) &= \langle\,\rangle \\ \mathsf{diff}_2(a \cdot s_1', s_2) &= \begin{cases} a \cdot \mathsf{diff}_2(s_1', s_2) & \text{se } a \notin s_2 \\ \mathsf{diff}_2(s_1', s_2) & \text{se } a \in s_2 \end{cases} \end{aligned}$$

Chiaramente, per implementare le equazioni ricorsive scritte sopra, è neccessario implementare una funzione ausiliaria `belongsTo` per testare se un certo valore appartiene o meno ad una sequenza (le condizioni $a \in s_2$ o $a \notin s_2$).

```
int belongsTo(lista L, int x){
   if (isEmpty(L)) return 0;
   if (head(L)==x) return 1;
   return belongsTo(tail(L));
}

lista diffFun(lista L, lista M){
   if (isEmpty(L)) return NULL;
   if (belongsTo(M, head(L)))
       return diffFun(tail(L), M);
   return cons(diffFun(tail(L), M), head(L));
}
```

Eliminazione dei Duplicati da una Lista. Concludiamo questa sezione, ponendoci un ultimo problema simile ai precedenti: data una lista, eliminare eventuali ripetizioni di elementi. Procediamo come al solito.

$$\begin{aligned} \mathsf{elimDupl}(\langle\,\rangle) &= \langle\,\rangle \\ \mathsf{elimDupl}(a \cdot s) &= a \cdot \mathsf{elimDupl}(\mathsf{remove}(s, a)) \end{aligned}$$

La versione funzionale è presto scritta, ma perchè ho deciso di invocare `removeRec` in luogo di `removeFun`?

```
lista elimDuplFun(lista L){
   if (isEmpty(L)) return NULL;
   return cons(head(L), elimDuplFun(removeRec(tail(L), head(L))));
}
```

Con poco più sforzo, ecco la versione *in place*. L'algoritmo è sostanzialmente lo stesso, prendendo le dovute usuali precauzioni.

```
lista elimDuplRec(lista L){
   if (!L) return NULL;
   L->next = elimDuplRec(removeRec(L->next, L->val));
   return L;
}
```

3.7 Confronto tra Liste e Array

Siamo pronti per affrontare brevemente il confronto tra la struttura dati lista concatenata e la struttura dati array. La lista ha una natura essenzialmente *sequenziale* e quindi per accedere ad un elemento è necessario scorrere sequenzialmente la lista; gli elementi dell'array sono invece accessibili direttamente specificando un indice (provare a scrivere la ricerca binaria in una lista ordinata!). Per contro, la lista risulta più flessibile nella soluzione di molti problemi, perché non è necessario specificarne a priori la lunghezza e finchè c'è memoria libera è possibile allocare nuovi elementi. La lista presenta un certo overhead di occupazione di memoria dovuto alla necessità di memorizzare oltre ai campi informazione anche i campi puntatore.

A volte tuttavia, la lista permette un grande *risparmio di memoria*: immaginate per esempio di avere una relazione binaria. Abbiamo visto che si può rappresentare con una matrice. Tuttavia, se ogni elemento fosse in relazione con pochi altri elementi converrebbe rappresentare ad esempio la relazione con un array di liste, in cui per ciascun elemento esplicitamente elenco gli elementi con cui è in relazione.

Un ultimo vantaggio della lista dipende dal fatto che alcune operazioni (come rimuovere o inserire un elemento) necessitano solo lo spostamento di alcuni puntatori, mentre in un array le stesse operazioni necessitano di costosi spostamenti di tutti gli elementi che stanno a destra del punto in cui ho eliminato o inserito l'elemento. Occorrerà sempre tenere ben presenti questi elementi al fine di utilizzare la giusta struttura dati, tenendo conto della natura del problema e degli obiettivi prioritari (efficienza, occupazione di memoria, flessibilità ...).

Un utile esercizio potrebbe essere quello di reimplementare gli algoritmi di ordinamento visti per gli array sulle liste. Alcuni, come *quickSort*, vi obbligheranno

a usare un po' di fantasia per implementarli sulle liste e probabilmente perderanno un po' in efficienza. Altri, come *insertionSort* o *mergeSort*, pur mantenendo la loro complessità asintotica, potrebbero trarre giovamento dal fatto di operare su liste piuttosto che su array.

Esercizi e Spunti di Riflessione

1. Dare le equazioni ricorsive che definiscono una funzione $\mathsf{filtra}(\mathsf{s}_1, \mathsf{s}_2)$ che restituisce una lista con tutti gli elementi che stanno sia in s_1 che in s_2. Scrivere poi le funzioni C `filtraFun`, `filtraRec` e `filtraIt`.

2. ★ Scrivere una procedura iterativa che stampa una lista rovesciata.

3. Scrivere una procedura iterativa che rovescia una lista in place.

4. Un elemento di una lista è *massimo locale*, se è maggiore del suo predecessore e del suo successore (per il primo è sufficiente che sia maggiore del successore e per l'ultimo è sufficiente che sia maggiore del predecessore). Scrivere una funzione ricorsiva e una iterativa che data in ingresso una lista L, restituiscano come risultato una *nuova* lista che contiene tutti i massimi locali di L.

5. Scrivere tre funzioni `removeFirstFun`, `removeFirstRec` e `removeFirstIt` che rimuovono solo la *prima* occorrenza di un certo valore x da una lista L. Ispirarsi alle funzioni `removeFun`, `removeRec` e `removeIt` che eliminano tutte le occorrenze di un valore.

6. ★ Scrivere tre funzioni `removeLastFun`, `removeLastRec` e `removeLastIt` che rimuovono solo l'*ultima* occorrenza di un certo valore x da una lista L. Scrivere funzioni di complessità *lineare* nella lunghezza della lista [SUGG: sfruttare nelle versioni ricorsive, il ritorno dalle chiamate!].

7. Se avete capito l'esercizio precedente, non dovreste avere difficoltà a scrivere una funzione che sostituisce il valore di ogni nodo con la somma degli elementi *successivi*. Esempio: data la lista $\langle 8, 11, -4, 5\rangle$ deve *modificare* la lista in modo che rappresenti la lista $\langle 20, 12, 1, 5\rangle$. Scrivere una funzione che sostituisce il valore di ogni nodo con la somma degli elementi *precedenti*. Esempio: data la lista $\langle 8, 11, -4, 5\rangle$ deve *modificare* la lista in modo che rappresenti la lista $\langle 8, 19, 15, 20\rangle$. Di entrambi i problemi, dare le versioni ricorsive e iterative. Discutere vantaggi e svantaggi dell'approccio ricorsivo rispetto a quello iterativo.

8. Scrivere tre funzioni `removeLastFun`, `removeLastRec` e `removeLastIt` che rimuovono solo l'*ultima* occorrenza di un certo valore x da una lista L. Scrivere funzioni di complessità *lineare* nella lunghezza della lista [SUGG: sfruttare nelle versioni ricorsive, il ritorno dalle chiamate!].

9. Considerate le seguenti equazioni ricorsive per specificare il comportamento della funzione append:

$$\begin{aligned} \mathsf{append}_2(s_1, \langle\,\rangle) &= s_1 \\ \mathsf{append}_2(s_1, a \cdot s_2') &= \mathsf{append}(\mathsf{addTail}(s_1, a), s_2') \end{aligned}$$

Scrivere i programmi C che implementano questa specifica. Perchè il programma scritto nella sezione 3 ha da ritenersi preferibile?

10. ♣ Dimostrare per induzione che le equazioni ricorsive dell'esercizio precedente specificano la stessa funzione `append` definita precedentemente nella sezione 3.

11. ♣ Cosa accade quando implemento una funzione come `reverseFun` chiamando la funzione `addTFun`? Oppure se definisco la funzione `elimDuplFun` chiamando `removeFun` invece di `removeRec`? Aiutarsi disegnando gli stati della memoria su un esempio concreto (con una lista di 3 o 4 elementi).

12. Una sequenza s' è *sotto-sequenza* della sequenza s se e solo se esistono due sequenze s_1, s_2 (eventualmente vuote) tali che $s = s_1 \cdot s' \cdot s_2$. (qui abusiamo dell'operatore $\cdot$ per denotare anche la concatenazione di sequenze). (a) Dare una definizione induttiva della relazione di sottosequenza mediante equazioni ricorsive. (b) Scrivere poi una funzione C `int subSeq(lista L, lista M)` che restituisca 1 se la lista `L` rappresenta una sotto-sequenza della sequenza rappresentata dalla lista `M`.

13. Una sequenza s_1 è *immersa* in s_2 se gli elementi di s_1 occorrono ordinatamente in s_2. (a) Dare una definizione induttiva della relazione di immersione mediante equazioni ricorsive. (b) Scrivere poi una funzione C `int immersa(lista L, lista M)` che restituisca 1 se la lista rappresentata da `L` è immersa nella sequenza rappresentata dalla lista `M`.

14. Scrivere una funzione C che restituisce 1 se due liste contengono gli stessi elementi (eventualmente con un numero di occorrenze diverso e disposti in ordine diverso) e 0 altrimenti.

15. Scrivere una funzione C che restituisce 1 se due liste contengono gli stessi elementi (eventualmente in ordine diverso, ma ciascuno con lo stesso numero di occorrenze) e 0 altrimenti.

16. ★ Implementare il tipo di dato *numero naturale*, in modo che il numero naturale n sia rappresentato da una lista con n nodi (i nodi contengono solo il campo puntatore). Scrivere prima le funzione base (successore, predecessore, test di zero), poi le usuali funzioni aritmetiche (somma, sottrazione, moltiplicazione, divisione e resto intero) ed infine esponenziale e fattoriale.

17. Provare a scrivere gli algoritmi di ordinamento *insertionSort*, *bubbleSort*, *quickSort* e *mergeSort* sulle liste, mantenendovi (per quanto possibile) aderenti alle versioni sui vettori. Dire quali algoritmi si giovano della rappresentazione a lista, quali soffrono e quali ne risultano un po' snaturati.

18. Scrivere una funzione che preso in input una lista di interi L, produca in output una *lista di liste* di interi, ciascuna contenente una sotto-sequenza crescente di L. Definire preventivamente in modo opportuno il tipo di dato lista di liste di interi.