ADT LISTA

Pochi Linguaggi forniscono tipo ***lista*** fra predefiniti (LISP, Prolog); per gli altri, ***ADT lista si costruisce a partire da altre strutture dati*** (in C tipicamente vettori o puntatori)

OPERAZIONI PRIMITIVE DA REALIZZARE

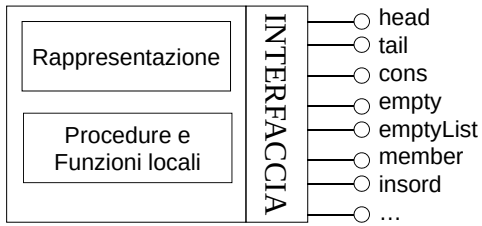
Operazione	Descrizione
cons: D x list -> list	Costruisce una nuova lista, aggiungendo l'elemento fornito in testa alla lista data
head: list -> D	Restituisce primo elemento della lista data
tail: list -> list	Restituisce la coda della lista data
emptyList: -> list	Restituisce (costruisce) la lista vuota
empty: list -> boolean	Restituisce vero se la lista data è vuota, falso altrimenti

ADT LISTA: altre operazioni non primitive

Operazione	Descrizione
member: D x list -> boolean	Restituisce vero o falso a seconda se l'elemento dato è presente nella lista data
length: list -> int	Calcola il numero di elementi della lista data
append: list x list -> list	Restituisce una lista che è concatenamento delle due liste date
reverse: list -> list	Restituisce una lista che è l'inverso della lista data
copy: list -> list	Restituisce una lista che è copia della lista data
insord: D x list -> list	Inserimento ordinato di un elemento del dominio D in una lista ordinata

COSTRUZIONE ADT LISTA

Incapsulare la ***rappresentazione concreta*** (che utilizza puntatori e strutture) e esportare sotto forma di file header, solo ***definizioni di tipo*** e ***dichiarazioni delle operazioni***



Funzionamento di lista ***non dipende dal tipo*** degli elementi di cui è composta -> ***soluzione generale***

COSTRUZIONE ADT LISTA (2)

LINEE GUIDA:

- definire un tipo ***element*** per rappresentare generico tipo di elemento (con le sue proprietà)
- realizzare ADT lista in termini di element

Il tipo element

File element.h contiene la definizione di tipo:

```
typedef int element;
```

(il file element.c non è necessario per ora)

Inoltre: `typedef enum { false, true } boolean;`

ADT LISTA

FILE HEADER (list.h)

```
#include "element.h"

typedef struct list_element {
       element value;
       struct list_element *next;
} item;
typedef item *list;

list emptyList(void);              // PRIMITIVE
boolean empty(list);
element head(list);
list tail(list);
list cons(element, list);

void showList(list);               // NON PRIMITIVE
boolean member(element, list);
…
```


ADT LISTA: file di implementazione (list.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"          /* ---- PRIMITIVE ---- */

list  emptyList(void)      { return NULL; }

boolean  empty(list l) {
       if (l==NULL) return true; else return false; }

element  head(list l) {
       if (empty(l)) abort();
       else return  l->value; }

list  tail(list l) {
       if (empty(l)) abort();
       else return l->next; }

list  cons(element e, list l) {
       list t;
       t = (list) malloc(sizeof(item));
       t->value=e; t->next=l; return t; }
```


ADT LISTA: file di implementazione (list.c)

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
...

void showList(list l) {              // NON PRIMITIVE
      printf("[");
      while (!empty(l)) {
            printf("%d", head(l));
            l = tail(l);
            if (!empty(l)) printf(", ");
      } printf("]\n");
}
```

NOTA: **`printf("%d", …)`** è specifica per gli interi

ESERCIZIO 5 (segue)

insord iterativa

```
list insord(element el, list l) {
  list pprec, patt = l, paux;
  boolean trovato = 0;
  while (patt!=NULL && !trovato) {
      if (el < patt->value) trovato = 1;
      else { pprec = patt; patt = patt->next; }
  }
  paux = (list) malloc(sizeof(item));
  paux->value = el; paux->next = patt;
  if (patt==l) return paux;
  else { pprec->next = paux; return l; }
}
```

ESERCIZIO 5 (segue)

insord iterativa, con funzioni primitive

```
list insord(int el, list l)
{   list pprec, patt = l, paux;
    int trovato = 0;

    while (!empty(patt) && !trovato){
        if (el < head(patt))   trovato = 1;
        else {  pprec = patt;
                patt = tail(patt); }   }
    paux=cons(el, patt);
    if (patt==l)  return paux;
    else {  pprec->next = paux;
            return l;   }
}
```

ADT LISTA: il cliente (main.c)

```
#include <stdio.h>
#include "list.h"

main() {
list l1 = emptyList();
int el;
do {  printf("\n Introdurre valore:\t");
      scanf("%d", &el);
      l1 = cons(el, l1);
      } while (el!=0);        // condizione arbitraria

showList(l1);
}
```

IL PROBLEMA DELLA GENERICITÀ

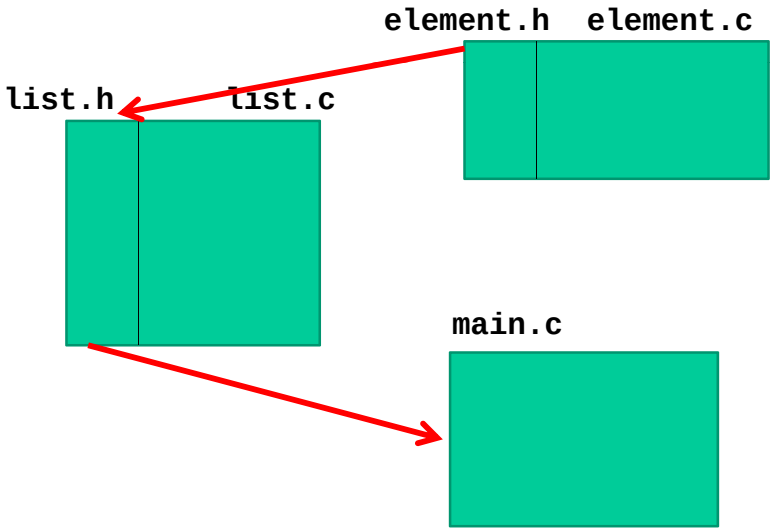
Funzionamento lista ***non deve dipendere dal tipo degli elementi*** di cui è composta => cercare di costruire ADT generico che funzioni con ***qualunque tipo di elementi***

=> ADT ausiliario ***element*** e realizzazione dell'ADT lista in termini di element

Osservazioni:
- ***showList*** dipende da printf() che svela il tipo dell'elemento
- ***insord*** dipende dal tipo dell'elemento nel momento del confronto
- …

Può quindi essere utile ***generalizzare queste necessità***, e definire un ADT element che fornisca funzioni per:
- verificare ***relazione d'ordine*** fra due elementi
- verificare ***l'uguaglianza*** fra due elementi
- leggere da ***input*** un elemento
- scrivere su ***output*** un elemento

COMPONENTI



ADT ELEMENT: element.h

Header element.h deve contenere
- ***definizione*** del tipo element
- ***dichiarazioni*** delle varie funzioni fornite

Poiché contiene una ***definizione***, header dovrà essere protetto dal ***problema delle inclusioni multiple***

```
#ifndef ELEMENT_H
#define ELEMENT_H
typedef int element;                    //DEFINIZIONI
typedef enum { false, true } boolean;

boolean isLess(element, element);       //DICHIARAZIONI
boolean isEqual(element, element);
element getElement(void);
void printElement(element);
#endif
```

TO DO

Si ***definiscano i file element.c e element.h*** che realizzano l'ADT ***element*** (come intero)

Si modifichino i file ***list.h e list.c già disponibili***, generalizzando le loro operazioni in funzione di quelle esportate dall'ADT element

Il ***main*** da realizzare deve leggere la sequenza e inserire ogni elemento letto in una ***lista ordinata*** e infine stampare la lista creata usando le funzionalità dell'ADT ***element*** e ***list***

COSA CAMBIA NELL'ADT LISTA?

Ridefinendo in funzione delle operazioni esportate da element.h il codice delle operazioni dell'ADT lista cerchiamo di aumentarne la riusabilità

Ad esempio, prima …

```
void showList(list l) {                 // NON PRIMITIVE
      printf("[");
      while (!empty(l)) {
            printf("%d", head(l));
            l = tail(l);
            if (!empty(l)) printf(", ");
      } printf("]\n");
}
```

NOTA: **`printf("%d", …)`** è specifica per gli interi

ADT list.c *prima …*

insord iterativa

```
list insord(element el, list l) {
  list pprec, patt = l, paux;
  boolean trovato = 0;
  while (patt!=NULL && !trovato) {
      if (el < patt->value) trovato = 1;
      else { pprec = patt; patt = patt->next; }
  }
  paux = (list) malloc(sizeof(item));
  paux->value = el; paux->next = patt;
  if (patt==l) return paux;
  else { pprec->next = paux; return l; }
}
```

DO IT!

Realizzare ***l'ADT element*** (intero e operazioni, si veda ***element.h***)

Modificare ***list.h e list.c già disponibili***

Il ***main*** da realizzare deve leggere la sequenza e inserire ogni elemento letto in una ***lista ordinata*** e stamparla

...

(vediamo poi la soluzione insieme)

GENERALIZZIAMO ULTERIORMENTE …

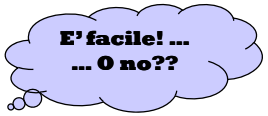
Un file sequenziale di tipo testo (ESPR.TXT) contiene stringhe (una per linea) costituite dai caratteri {a,b,c,*,+}. Si realizzi un programma C che:

a) riconosca le stringhe del file uguali a "a*b+c" oppure a "a+b*c".

b) inserisca le stringhe riconosciute e i loro numeri di linea in una lista a puntatori, L1, ***ordinata sul campo stringa (a parità di stringa, si ordini in base alla posizione)***;

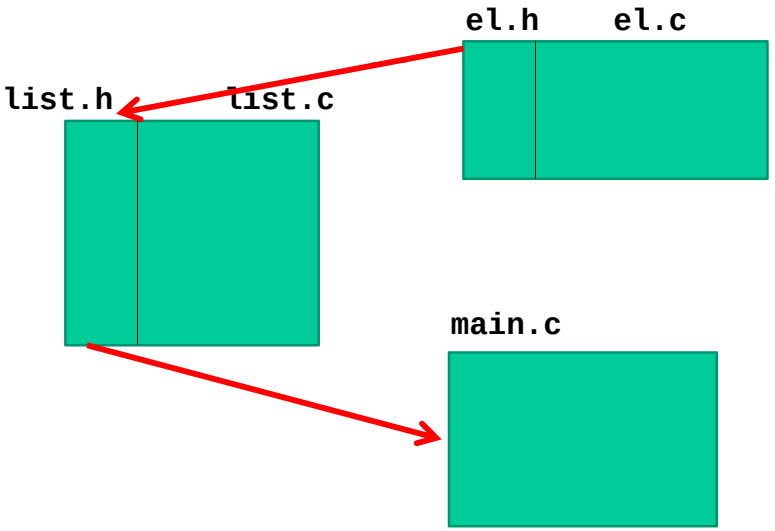
c) produca, a partire dalla lista L1 generata precedentemente, un file (UNICHE.TXT) di tipo testo in cui ogni stringa accettata compare seguita, sulla stessa linea, dalla posizione originale nel file ESPR.TXT

Esempio:



ESPR.TXT:	UNICHE:	
a*b+c	a*b+c	1
a**b+	a*b+c	4
a+b*c	a+b*c	3
a*b+c	a+b*c	5
a+b*c		

COMPONENTI



Interfaccia modulo elementi: el.h

```
/* ELEMENT TYPE - file el.h*/
#ifndef ELEMENT_H
#define ELEMENT_H

#define N 30
typedef struct { char stringa[N];
                 int  pos; }  element;
typedef enum {false, true} boolean;

boolean isLess(element, element);
boolean isEqual(element, element);
element getElement(char *, int);
void printElement(FILE *,element);
#endif
```

TO DO: el.c

Realizzare la parte implementazione del nuovo ADT degli elementi (strutture)

`el.h` `el.c`

Ove necessario rivedere le funzioni dell'ADT lista

`list.h` `list.c`

…

(vediamo poi la soluzione insieme)

TO DO … MORE

c) produca, a partire dalla lista L1 generata precedentemente, un file (UNICHE.TXT) di tipo testo in cui ogni stringa accettata compare una sola volta seguita, sulla stessa linea, dalle posizioni originali nel file ESPR.TXT

Esempio:

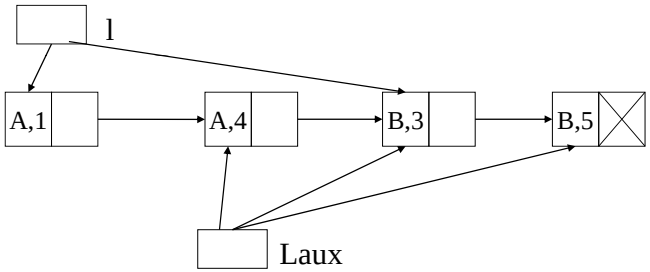
ESPR.TXT:

a*b+c
a**b+
a+b*c
a*b+c
a+b*c

UNICHE:

a*b+c 1 4
a+b*c 3 5

Come scandire la lista?



A 1 4

B 3 5

Scansione innestata, avanzo con Laux (che parte da tail(l)) finché trovo nodi con la stessa stringa (o arrivo alla fine della lista); non appena trovo una stringa diversa, avanzo con l fino a questo nodo e riprendo la scansione innestata (partendo con Laux=tail(l)).

REALIZZATE QUESTA FUNZIONE

```
void fshowList2(FILE *f, list l);
```

CONCLUSIONE

Che cosa possiamo trarre come ulteriore conclusione?