Tipi di Dato

Tipi di Dato (data types)

Data Object (Dato) $\Longleftrightarrow$ contenitore per valori

valori=pattern di bit, interpretabili come:

- numeri
- caratteri
- $\cdots$

data object $\neq$ valore

(a volte sono identificati)

Dato $\leftarrow$ Tempo di Vita

- elementare: valori manipolati come unità
- strutturato ("struttura dati"): aggregato di altri dati (data objects)

Dato:

- insieme **attributi**: invarianti durante il tempo di vita
  - tipo
  - numero dei possibili valori
  - organizzazione logica dei valori
  - ...

- insieme di **legami**: possono variare durante il tempo di vita
  - tipo $\leftrightarrow$ compilatore (insieme dei possibili valori che il dato può assumere
  - locazione $\leftrightarrow$ gestore memoria (run-time; locazioni di memoria)
  - valore $\leftrightarrow$ assegnamento
  - nome $\leftrightarrow$ dichiarazione (compilazione)/ chiamate sottoprogramma (run-time)
  - componente $\leftrightarrow$ modifica puntatori (il legame di un dato ad uno o più dati di cui esso è componente è spesso rappresentato da un puntatore, e può essere modificato attraverso il cambiamento/la modifica del puntatore medesimo)

Variabili e Costanti

- **variabile**: dato definito con un nome in un programma
- **variabile semplice**: il dato associato è elementare
- **costante**: dato con nome il cui valore è costante durante il tempo di vita
- **letterale**: costante il cui nome è una rappresentazione del valore
- **costante utente**: costante il cui nome è scelto dall'utente

Esempio:

```
#define pp -256
...
int x;
int y = 4;
const int z = 7;
...
if (x==((y+z)/20+pp)) ···
```

Tipo di Dato

Un tipo di dato è una coppia

$$< D, O >$$

dove D è una classe di data object e O è un insieme di operazioni per crearli e manipolarli.

Linguaggio L:

- insieme di tipi di dato primitivi
- meccanismi per la definizione di nuovi tipi di dato
- meccanismi per manipolare tipi (nuovi linguaggi)

Tipo di Dato

Specifica

- indipendentemente dall'implementazione
  - tipo dei dati
  - operazioni
- dipendente dall'implementazione
  - domini semantici (valori possibili)

Implementazione

simulazione di una parte della macchina astratta

- rappresentazione concreta dei dati
- rappresentazione algoritmica delle operazioni come procedure

Rappresentazione Sintattica

poco interessante

Specifica dei TdD Elementari

Un data object elementare contiene un singolo data value. Una classe di tali dati su cui vengono definite delle operazioni è detta **elementary data type**.

attributi: gli attributi di base di un qualsiasi data object sono solitamente invarianti durante il ciclo di vita (es. tipo, nome). Alcuni attributi possono essere memorizzati in un **descrittore** durante l'esecuzione; altri possono essere usati solo per determinare la rappresentazione in memoria del dato e non apparire esplicitamente durante l'esecuzione.

valori: il tipo del dato determina l'insieme dei possibili valori che esso può contenere. Tale insieme è tipicamente un insieme ordinato con `min` e `max`.

operazioni:

- primitive: fanno parte della definizione di L
- definite dal programmatore: sottoprogrammi

Operazioni

desiderio: `operazioni = funzioni`

tipo dell'operazione ⇒ segnatura (signature)

numero, ordine, tipo degli argomenti nel dominio dell'operazione

\+

tipo del risultato

Notazione matematica per la specifica delle operazioni:

$\texttt{nome op} : \texttt{tipo}_1 \times \cdots \times \texttt{tipo}_n \rightarrow \texttt{tipo}$

- alcuni linguaggi codificano direttamente il concetto di segnatura (C,C++):

  $\mathtt{sum} : \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int}$

  $\mathtt{int\ sum(int, int);} \qquad /*\ \mathtt{prototipo}\ */$

- altri linguaggi inferiscono automaticamente la segnatura (tipo) (ML, CAML):

  $\mathtt{sum} : \mathtt{int} * \mathtt{int} \rightarrow \mathtt{int}$

  (questa è la risposta del sistema)

- altri linguaggi codificano implicitamente la segnatura delle operazioni utente (Pascal, Fortran):

  $\mathtt{function\ sum(x : integer; y : integer) : integer;}$
  $< \mathtt{codice\ della\ funzione} >$

desiderio: `operazioni = funzioni`

Le difficoltà nella sua realizzazione sono:

- operazioni indefinite per certi input: può essere estremamente difficile specificare l'esatto dominio su cui l'operazione è indefinita

- argomenti impliciti (variabili globali o altri riferimenti non locali): la determinazione completa di tutti i dati che possono influenzare il risultato può essere complessa

- effetti collaterali (side effects): risultati impliciti, ovvero modifica del valore di dati, definiti sia dall'utente sia dal sistema (gli effetti collaterali sono comunque la base per alcune operazioni, quale l'assegnamento)

- auto-modifiche (history sensitivity): il risultato di un'operazione può dipendere non solo dagli argomenti, ma anche da tutta la storia delle sue precedenti chiamate (es. generatore di numeri casuali)

Implementazione dei TdD Elementari

Rappresentazione in memoria

Tipicamente basata sull'HW della macchina sottostante. Altrimenti le operazioni vanno simulate via SW.

Attributi:

- determinati dal compilatore, non memorizzati nella rappresentazione run-time (efficienza; C, Fortran, Pascal)

- memorizzati in un descrittore, presente come parte del data object durante il run-time (flessibilità; Lisp, Prolog)

Implementazione delle operazioni

- diretta, come operazioni HW

- sottoprogrammi

- sequenze di codice in-line

Dichiarazioni

Lo scopo delle dichiarazioni è quello di fornire al compilatore informazioni su dati e programmi.

Dichiarazioni di Dati

Scopo: type-checking statico.

Non tutti i L hanno la dichiarazione di dati

- in certi L è obbligatorio dichiarare tutti gli identificatori:

  `C :   int x, y;`

- in altri L (Lisp, Prolog) le dichiarazioni non hanno senso in quanto sono linguaggi type-free

- in altri ancora (ML) è la macchina che cerca di inferire i tipi

Esempio: "somma" in ML

$> \mathtt{fun\ somma(x:\ int,\ y:\ int)\ =\ (x+y):\ int;}$

$> \mathtt{fun\ somma(x:\ int,\ y:\ int)\ =\ (x+y);}$

$> \mathtt{fun\ somma(x:\ int,\ y)\ =\ (x+y);}$

$> \mathtt{fun\ somma(x,\ y)\ =\ (x+y);}$
$(*\ \mathtt{se} \geq \mathtt{ML}_{97}\ \mathtt{altrimenti\ error}\ *)$

$- \mathtt{somma} : \mathtt{int} * \mathtt{int} \to \mathtt{int}$

Dichiarazione di Operazioni

L'informazione necessaria durante la traduzione è essenzialmente la segnatura dell'operazione.

Non è richiesta la dichiarazione esplicita delle operazioni primitive.

$\mathtt{minus} : \ \mathtt{int} \rightarrow \mathtt{int}$

$\mathtt{C} :$

$\mathtt{int\,minus\,(int);} \quad /*\ \mathtt{dichiarazione}\ */$

$\mathtt{int\,minus\,(int)}$

$\{\mathtt{return\,x} - 1;\} \quad /*\ \mathtt{definizione}\ */$

$\mathtt{ML} :$

$> \mathtt{fun\,minus(x} : \ \mathtt{int)} \ = \ \mathtt{x} - 1;$

$- \mathtt{minus} : \mathtt{int} \rightarrow \mathtt{int}$

Type Checking

In memoria

$$\texttt{001101...110}$$

cosa rappresenta? Non si può dire: potrebbe essere un intero, un reale, un indirizzo, un'istruzione ...

A livello di operazioni primitive HW, non viene eseguito alcun controllo su ciò che rappresentano effettivamente gli operandi.

Type checking significa controllare che ciascuna operazione eseguita da un programma riceva il numero appropriato di argomenti, e del tipo atteso.

Pascal:

x:=y+z;

+: num $\times$ num $\rightarrow$ num

dove num = indifferentemente intero o reale

$\Rightarrow$ NO, non ha senso

(a livello HW le due somme sono diverse!)

`+:  int` $\times$ `int` $\rightarrow$ `int`

`+:  real` $\times$ `real` $\rightarrow$ `real`

$\Rightarrow$ `fenomeno dell'overloading`

Il type checker assicura dunque che il tipo degli argomenti delle operazioni sia corretto; può lavorare:

- run time (type checking dinamico)
  - vantaggio: flessibilità (il significato di A+B può cambiare nel corso della computazione)
  - svantaggi
    - difficoltà di debugging (bisognerebbe provare tutti i cammini di esecuzione)
    - richiesta di memoria supplementare durante l'esecuzione
    - dato che l'implementazione del type checking è SW, la velocità di esecuzione ne risente

- compile time (type checking statico): vantaggi e svantaggi sono complementari a quelli del controllo dinamico. Durante la traduzione, le info memorizzate nella symbol table sono:

- per ciascuna operazione, numero, ordine e tipi di argomenti e risultato
- per ciascuna variabile, il tipo del dato riferito col nome
- per ciascuna costante, il tipo del dato

Il type checking dinamico è necessario nei linguaggi type free (Lisp, Prolog) nei quali non esiste il concetto di tipo.

`Esempio C/C++:` $E_1 + E_2$

`type check:`

`tipo(`E_1`) = tipo(`E_2`) = int` $\Rightarrow$ `OK`

`tipo(`E_1`) = tipo(`E_2`) = real` $\Rightarrow$ `OK`

`tipo(`E_1`)` $\neq$ `tipo(`E_2`)` $\Rightarrow$ `Type Mismatch`

Type Mismatch

Se durante il type checking viene rilevato un mismatch, due sono le strade percorribili:

- viene segnalato un **errore**
- viene effettuata una **conversione implicita di tipo** (detta anche **coercion**) (C, Pascal)

Il principio guida delle conversioni implicite è che esse non devono provocare perdita di informazione. Se ad esempio:

$$E_1 + E_2$$

$\texttt{tipo(}E_1\texttt{) = int e tipo(}E_2\texttt{) = real}$

allora E_1 viene convertito in `real`.

Una conversione di tipo è un'operazione con segnatura:

$$\texttt{conversione} : \texttt{tipo}_1 \rightarrow \texttt{tipo}_2$$

In molti linguaggi esiste un insieme di funzioni che il programmatore può esplicitamente invocare per effettuare conversioni di tipo. Ad esempio il **cast** in C/C++:

```
(int)3.4
```

dove l'`(int)` è l'operatore di conversione. È chiaro (come nell'esempio precedente) che la conversione di tipo esplicita può far perdere informazione.

La coercion può essere

- dinamica: type checking dinamico
- statica: type checking statico (aggiunto codice)

Tipicamente nei linguaggi, il type checking statico non è sempre possibile: possono esserci alcuni costrutti, in particolari condizioni, per i quali non può essere effettuato. Per trattare questi casi particolari:

- type checking dinamico: costo elevato
- si lascia l'operazione senza controllo

Se in un programma è possibile individuare staticamente tutti gli errori di tipo, allora il linguaggio è detto **tipizzato forte** (**strongly typed**).

Inferenza di Tipi

Nei linguaggi funzionali tipati, spesso è presente un meccanismo di inferenza dei tipi, ovvero il type checker inferisce automaticamente i tipi.

L'idea è quella di non dichiarare esplicitamente i tipi (se possibile).

Esempio ML:

```
> fun succ(x: int) = x+1;

- succ: int → int

> fun rsucc(x: real) = x+1.0;

- succ: real → real
```

Ad esempio in `x+1`, `1` è `int`, l'unica operazione di somma applicabile è la $+$: `int → int`, pertanto `x` deve essere di tipo `int`. Infatti, si può anche scrivere

```
> fun succ(x) = x+1;
```

e la segnatura della funzione viene inferita automaticamente e nella maniera attesa.

Assegnamento

L'assegnamento è l'operazione di base per cambiare il legame (binding) tra un dato e il suo valore.

È un'operazione che interessa essenzialmente i tipi di dato elementari, e funziona per `effetto collaterale (side effect)` sulla memoria.

`C:`

`(=):` $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$, o meglio

`(=):` $\texttt{int} \times \texttt{int} \times \texttt{store} \rightarrow \texttt{int} \times \texttt{store}$

`(con gli store impliciti)`

`Pascal:`

`(:=):` $\texttt{int} \times \texttt{int} \rightarrow \texttt{void}$

Assegnamento

$$X \leftarrow E$$

dove

$\leftarrow$: operatore di assegnamento

X: **l-value** $\Rightarrow$ indirizzo dell'identificatore

E: **r-value** $\Rightarrow$ valore memorizzabile

Definizione di assegnamento:

1. calcola l-value $\Rightarrow$ valore lv
2. calcola r-value $\Rightarrow$ valore rv
3. assegna rv a lv
4. restituisci rv

Inizializzazione

Il valore che assume un data object (variabile), immediatamente dopo esser stato creato, dipende dall'inizializzazione, che può essere

- non prevista (Pascal)
- facoltativa (Ada, C)
- obbligatoria (APL)

È possibile che venga assegnato un valore `nullo` alle variabili dichiarate senza inizializzazione, ma spesso dipende dall'implementazione.

È buona regola inizializzare sempre esplicitamente (o con un assegnamento) tutte le variabili.

Principali TdD Elementari

Numeri Interi

Specifica

- un data object di tipo intero non ha tipicamente altri attributi oltre il suo tipo (nome). L'insieme dei valori forma un sottoinsieme ordinato e finito (`[minint,maxint]`) degli interi matematici.
- operazioni
  - aritmetiche

    $+ : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$

    $* : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$

    ...
  - relazionali

    $<: \texttt{int} \times \texttt{int} \rightarrow \texttt{bool}$

    ...

- (continuazione operazioni)
  - assegnamento

    $\leftarrow$: $\texttt{int} \times \texttt{int} \rightarrow \texttt{void}$

    $\leftarrow$: $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$
  - bitwise

    $\texttt{shift} : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}$

    ...
  - costanti

    $\texttt{maxint} : \texttt{void} \rightarrow \texttt{int}$

    $\texttt{minint} : \texttt{void} \rightarrow \texttt{int}$

Esempi:

C ha quattro tipi di dato: **`int, short, long, char`**

Pascal: **`integer`**

ML: **`int`**

Implementazione: se possibile è usato il tipo di dato corrispondente della macchina ospite.

Il numero binario è espresso in complemento a due

Numeri Reali (floating, virgola mobile)

Specifica

- tipo: float, (minimo,massimo)
- operazioni: $+, *, /, \ldots$

Implementazione: reali della macchina ospite

S	E	M
1 bit	8/11 bit	23/52 bit

Interpretazione

singola precisione:
$(-1)^S\ 1.M\ 2^{E-127}$, $E \in [0, 2^8 - 1] = [0, 255]$
se $E = 0$ e $M = 0 \Rightarrow$ num. rappresentato è lo 0

doppia precisione:
$(-1)^S\ 1.M\ 2^{E-1023}$, $E \in [0, 2^{11} - 1] = [0, 2047]$
se $E = 0$ e $M = 0 \Rightarrow$ num. rappresentato è lo 0

Altri Tipi Numerici

- numeri reali a virgola fissa
- numeri razionali (virtualmente a precisione infinita)
- numeri complessi
- subrange

  Esempio Pascal:

  `A: 1..10;`

  `1..10 è il tipo, sottotipo di int`

  $\Rightarrow$ `occorre type checking dinamico:`

  `A:=A+1;`

  `il valore (A+1)` $\in [1, 10]$

Tipi di Dato Enumerati

C: `enum Corsi {prog1,prog2,logica}` ($\{prog_1, prog_2, logica\}$)

Pascal: `type Corsi=(prog1,prog2,logica)` ($(prog_1, prog_2, logica)$)

Specifica

- tipo: nome + lista ordinata di elementi, $[val_1, \ldots, val_k]$
- operazioni: relazionali, assegnamento, ...

NB: ogni singolo tipo enumerato ha le proprie operazioni.

Implementazione: interi della macchina ospite

Booleani

Specifica

- tipo: bool + 2 valori (`V, F`)
- operazioni:

  `and` : `bool` $\times$ `bool` $\rightarrow$ `bool`

  `or` : `bool` $\times$ `bool` $\rightarrow$ `bool`

  ...

Implementazione: è sufficiente 1 bit, coi due valori binari

Esempi:

Pascal: `type boolean=(false,true);`

C: non esiste il tipo bool. Nelle espressioni logiche:

`falso` $\Rightarrow$ `numero intero 0`

`vero` $\Rightarrow$ `numero intero` $\neq$ `0`

Caratteri

char: C

string: ML, Prolog

Specifica

- tipo: nome + lista ordinata di caratteri ammessi, $[c_1, \ldots, c_k]$
- operazioni: confronto, assegnamento

Implementazione: quasi sempre appoggiata alla macchina ospite

- caratteri ASCII: la più usata (1 byte per carattere)
- caratteri EBCDIC
- $\ldots$

Tipo di Dato Strutturati

Un data object che è costruito come aggregato di altri data object (`componenti`), è detto TdD Strutturato, o Struttura Dati.

Specifica attributi

1. nome del tipo $\Rightarrow$ raro
2. tipi dei componenti
3. numero dei componenti
   - lunghezza fissa (array)
   - lunghezza variabile (stack)
4. nomi dei singoli componenti, ovvero meccanismi per l'accesso ai singoli elementi
5. numero massimo di componenti (se lunghezza variabile)
6. organizzazione dei componenti

Specifica operazioni

- selezione dei singoli componenti
- operazioni globali sulla struttura (es. assegnamento tra strutture in C)
- inserzione/cancellazione di elementi
- creazione/distruzione di strutture

Un tipo di dato per il quale le modifiche sono date dalle modifiche dei singoli componenti è detto `modificabile`.

Esempio: `int v[10]; ··· v[3]=exp;`

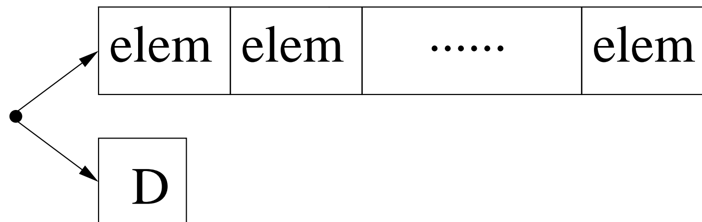
L'idea "sbagliata" è quella di avere

1. il tipo `array(int)`
2. l'operazione di assegnamento
   $=$: $\texttt{array(int)} \times \texttt{int} \times \texttt{int} \rightarrow \texttt{array(int)}$
   dove la prima coppia $\texttt{array(int)} \times \texttt{int}$ rappresenta l'l-value, mentre il rimanente `int` l'r-value.

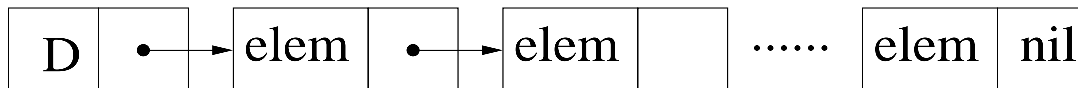
Normalmente, per motivi di efficienza, non è disponibile l'assegnamento tra array.

Implementazione: rappresentazione in **memoria**

- rappresentazione dei componenti
  - sequenziale



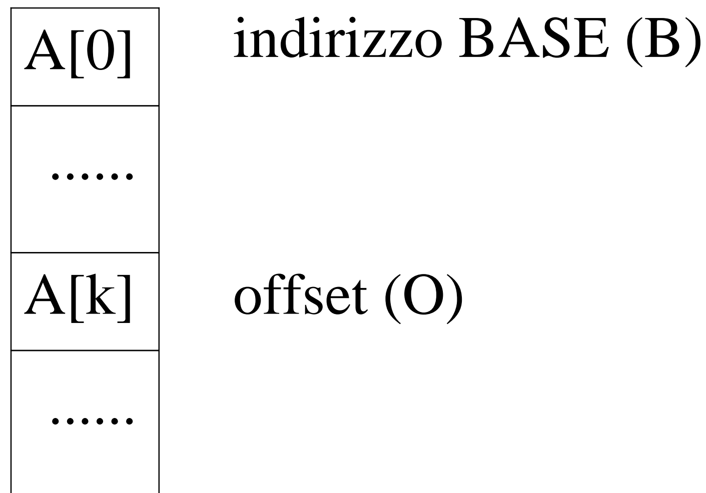
  - a liste



- descrittore (opzionale)

Implementazione operazioni

- rappresentazione sequenziale



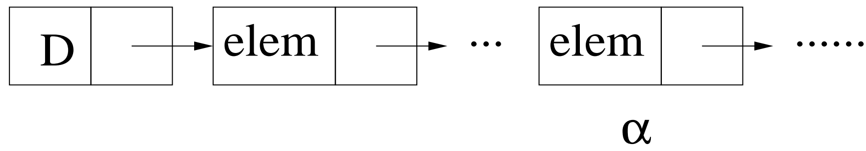

```
B = l-value(A) = l-value(A[0])
O = k×dim(elem)

l-value(A[k]) = indirizzo(A[k])=
= B+O = l-value(A) + k×dim(elem)
```

Osservazione:

`l-value(A[n])`

è calcolabile a compile-time se `n` è un letterale.

- rappresentazione a lista



Il calcolo di

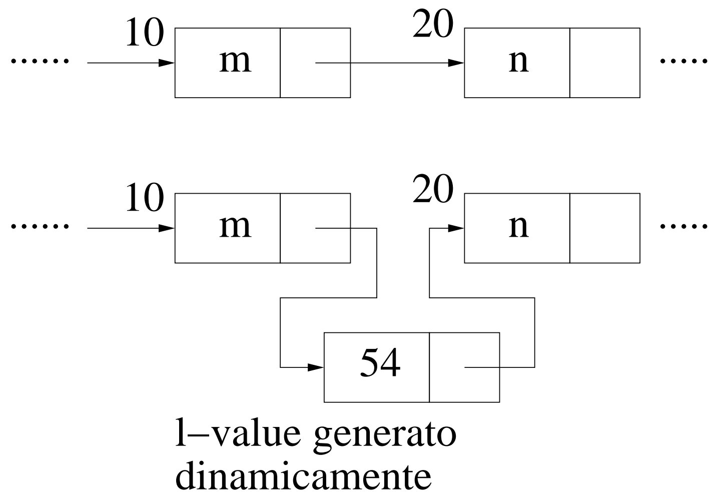
$$\texttt{l-value}(\alpha)$$

impone una scansione della lista.

L'implementazione di operazioni quali inserzione/cancellazione è semplice:

`insert` : `tipo` $\times$ `l-value` $\rightarrow$ `void` (side effect)

Esempio: `insert(54,10)`



Dichiarazioni e Type Checking

I concetti sottostanti alla dichiarazione di strutture dati non differiscono sostanzialmente da quelli dei TdD elementari. Tipicamente le dichiarazioni sono solo più complesse a causa del maggior numero di attributi da specificare.

Esempi:

$\texttt{Pascal}: \quad \texttt{B}: \texttt{array}[1..10, -7.. -1]\ \texttt{of}\ \texttt{real};$
$\texttt{C}: \qquad \texttt{float}\ \texttt{B}[10][7];$

$\texttt{B}[11][2] \rightarrow \texttt{errore}$

Senza type checking dinamico (es. C) rinunciamo a rilevare l'errore.

È assimilabile ad un errore di tipo.

Array

Gli array sono una struttura omogenea, e sono

- lineari (una dimensione; vettori)
- multidimensionali (matrici)

Vettori

Specifica attributi

- numero componenti
- tipo (unico!) dei componenti
- nomi dei singoli componenti $\leftrightarrow$ indice per selezionare elementi (campo di variabilità)

Specifica operazioni

- selezione elementi (calcolo l-value/r-value)
- creazione/distruzione

Implementazione: rappresentazione in **memoria**

Sia δ la lunghezza del descrittore (fissa).

Chiamiamo $\alpha = \beta + \delta$ (indirizzo della prima componente)

```
l-value(A[I]) =
= (β + δ)+(r-value(I)-LB)×E =
= (α - LB×E) + r-value(I)×E =
= k + r-value(I)×E
```
(k costante)

Fortran: `k` è nota a compile-time

Pascal: `k` è costante dopo che il vettore è stato allocato

Il valore `k` è detto **Inizio Virtuale** (VO)

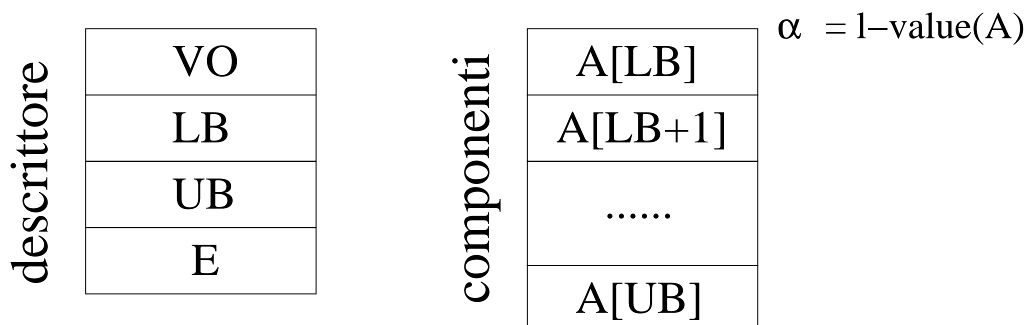
$$\texttt{k} = \alpha - \texttt{LB}\times\texttt{E} = \texttt{l-value(A[0])} = \texttt{VO}$$

è detto virtuale in quanto può non esistere (quando non esiste la componente 0).

Ora la formula di accesso alle componenti diventa:

$$\texttt{l-value(A[I])} = \texttt{VO} + \texttt{r-value(I)}\times\texttt{E}$$

Se VO è memorizzato nel descrittore, la contiguità tra esso e le componenti non è richiesta



Array Multidimensionali

Consideriamo prima il caso **bidimensionale**

Pascal: `B : array[3..10, 2..5] of real;`
→ `A[i, j]`

descrittore

VO
LB_1
UB_1
LB_2
UB_2
E

elementi (per righe)

α = l–value(A)

A[3,2]
A[3,3]
A[3,4]
A[3,5]
A[4,2]
......
A[10,2]
A[10,3]
A[10,4]
A[10,5]

Siano:

α = l-value(A)

S = dim di 1 riga = $((\mathtt{UB}_2 - \mathtt{LB}_2) + 1) \times \mathtt{E}$

VO = α - $\mathtt{LB}_1 \times \mathtt{S} - \mathtt{LB}_2 \times \mathtt{E}$

$\texttt{l-value(A[i,j])} =$
$= \alpha + (\texttt{i}-\texttt{LB}_1)\times\texttt{S} + (\texttt{j}-\texttt{LB}_2)\times\texttt{E} =$
$= \alpha + \texttt{i}\times\texttt{S} - \texttt{LB}_1\times\texttt{S} + \texttt{j}\times\texttt{E} - \texttt{LB}_2\times\texttt{E} =$
$= \texttt{VO} + \texttt{i}\times\texttt{S} + \texttt{j}\times\texttt{E}$

Nel caso del C:

$$\texttt{VO} = \alpha = \texttt{l-value(A[0][0])}$$

Nel caso **multidimensionale**:

Sia $\texttt{A}[\texttt{L}_1 : \texttt{U}_1, \ldots, \texttt{L}_n : \texttt{U}_n]$, vettore multidimensionale memorizzato a partire dall'indirizzo α.

- calcolo dei moltiplicatori $\texttt{m}_i$:
  $\texttt{m}_n = \texttt{E}$
  $\texttt{m}_i = (\texttt{U}_{i+1} - \texttt{L}_{i+1} + 1) \times \texttt{m}_{i+1} \qquad i \in [1, n-1]$

- calcolo dell'origine virtuale:
  $\texttt{VO} = \alpha - \sum_{i=1}^{n}(\texttt{L}_i \times \texttt{m}_i)$

- accesso alla singola componente:
  $\texttt{l-value}(\texttt{A}[\texttt{i}_1, \ldots, \texttt{i}_n]) = \texttt{VO} + \sum_{k=1}^{n} \texttt{i}_k \times \texttt{m}_k$

L'idea "logica" che sottende a questa implementazione è che

$$\mathtt{A[L_1 : U_1, \ldots, L_n : U_n]}$$

sia un vettore

$$\mathtt{A[L_1 : U_1]}$$

di elementi di tipo

$$\mathtt{[L_2 : U_2, \ldots, L_n : U_n]}$$

e che

$$\mathtt{A[i_1, \ldots, i_n]}$$

sia un'abbreviazione per

$$(\cdots((\mathtt{A[i_1]})[i_2])\cdots)[i_n]$$

Esaminiamo secondo questa interpretazione il caso bidimensionale. In questo caso

$$\mathtt{A[LB_1 : UB_1, LB_2 : UB_2]}$$

è interpretato come un vettore

$\mathtt{A[LB_1 : UB_1]}$ di elementi di tipo $\mathtt{[LB_2 : UB_2]}$

`l-value(A[i,j]) =`
`= l-value((A[i])[j])`

- `l-value(A[i]) =` α `+ (i-`LB_1`)`$\times$`S`
- `l-value((A[i])[j]) = l-value(B[j])`
  dove `B` è un vettore con
  `l-value(B) =` α_B `= l-value(A[i])`

per cui

`l-value(A[i,j]) =` α_B `+ j`$\times$`S =`
`=` α `+ (i-`LB_1`)`$\times$`S + j`$\times$`E`

L'idea appena espressa è un caso particolare delle **slices** (sottostrutture di array che a loro volta sono array).

Ad esempio in C dichiarare una variabile del genere (per qualche tipo `mytype`)

```
mytype A[m][n]
```

equivale a

1. `typedef mytype TYPEROW[n]`
2. `TYPEROW A[m]`

e pertanto scrivere

```
A[i][j]
```

equivale a scrivere

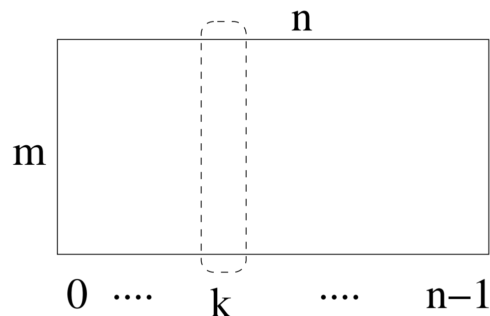
```
(A[i])[j]
```

Slices

Ad es. in un linguaggio C-like: `mytype A[m][n]`

VO	α
LB_1	ϕ
UB_1	m-1
$MULT_1$	n×sizeof(mytype)
LB_2	ϕ
UB_2	n-1
$MULT_2$	sizeof(mytype)

$$\texttt{l-value(A[i][j])} = \alpha + i \times MULT_1 + j \times MULT_2$$



slices C=A[_][k]

VO	α + (k×sizeof(mytype))
LB_1	ϕ
UB_1	m-1
$MULT_1$	n×sizeof(mytype)

$$\texttt{l-value(C[j])} = VO + j \times MULT_1$$

Record

È una struttura dati non omogenea. Le componenti, di tipo diverso, sono riferite con nomi simbolici.

```
struct nomestr
    {
      float a;
      int b;
      struct nomestr *p;
    }
```

attributi

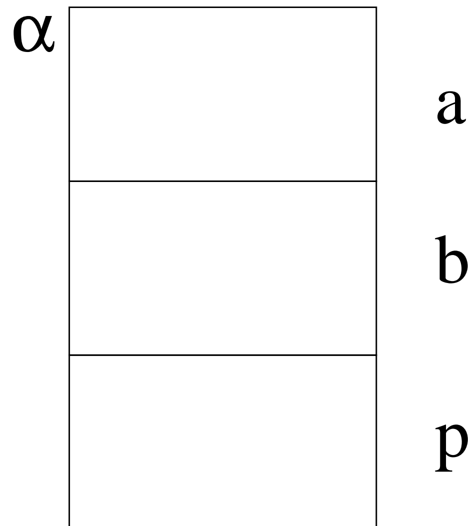
- numero componenti
- tipo di ciascun componente
- selettore usato per ogni componente

```
struct nomestr A;
struct nomestr B;

A.a = 3.62;
A.b = (int)A.a;
B=A;
```

implementazione: blocco contiguo di celle

→ usualmente non c'è un descrittore di record

→ ogni elemento può avere il suo descrittore



`A.nome` → meccanismo sintattico di accesso alle componenti.

Sia `nome` la componente i_esima:

`l-value(A.nome) =`

`= l-value(A.i) =`

`=` α `+` $\sum_{j=1}^{i-1}$ `(size of A.j) =` $\alpha + K_{nome}$

dove $K_{nome} = K_i \Rightarrow$ determinato a compile-time

Possono esservi dei problemi di **allineamento**:

supponiamo che:

`sizeof(int) = 4`
e che gli interi debbano avere indirizzi multipli di 4
e che ogni struttura sia allocata ad indirizzi multipli di 4

```
struct foo
        { char x;
          int y; } Z
```

⇒ (traduttore)

```
struct foo
        { char x;
          char dummy[3];
          int y; } Z
```

Record con Varianti

```
type A = (v,w);

var C : record
            uno: integer;
            case due: A of
                v: (tre: integer;
                    quattro: integer);
                w: (cinque: real;
                    sei: char)
            end;
```

componenti fisse: `uno` e `due`

se `C.due=v` ⇒ componenti: `tre` e `quattro`

se `C.due=w` ⇒ componenti: `cinque` e `sei`

La componente `due` è detto `tag` o `discriminante`

È possibile un check sul tag a run-time (Pascal, Ada)

implementazione: viene allocata la dimensione massima tenendo conto delle varianti

α

uno

due

tre | cinque

quattro | sei

`attenzione`: in caso di mancanza del tag (Pascal) possibili errori a run-time

Lista

Sequenza ordinata di strutture dati che sono

- eterogenee (Lisp)
- omogenee (ML)

1. **liste omogenee di tipo T**

   operazioni:

```
nil:  void → list(T)
cons:  T × list(T) → list(T)
head:  list(T) → T
tail:  list(T) → list(T)
```

nil : lista vuota (`[]`)
cons : costruttore di lista (`::`)
head/tail sono definite

Esempi ML:

```
[2,3,4] = 2::(3::(4::[]))
head([2,3,4]) = 2
tail([2,3,4]) = [3,4]
```

Un'operazione importante è il **pattern matching**

`pattern` ⇒ espressione costituita da variabili, costanti, costruttori, caratteri jolly

Esempio ML: consideriamo il pattern

`r::l`

- match con `[2,5,6]` ⇒ `r`↔`2` `l`↔`[5,6]`
- match con `[2]` ⇒ `r`↔`2` `l`↔`[]`

Una possibile definizione della funzione `head`:

```
fun head(x::_) = x
```

2. liste in un linguaggio senza tipi

`E` $\Rightarrow$ “tipo” di tutti gli oggetti sintattici del linguaggio

`LIST` $\Rightarrow$ “tipo” delle liste

nota: ovviamente `LIST` sottotipo di `E`

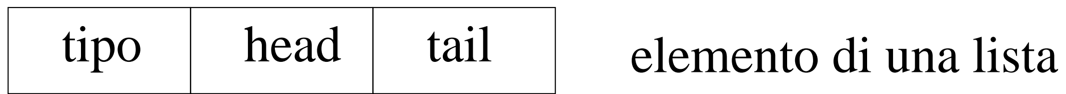
`cons:` $\quad$ `E` $\times$ `LIST` $\rightarrow$ `LIST`

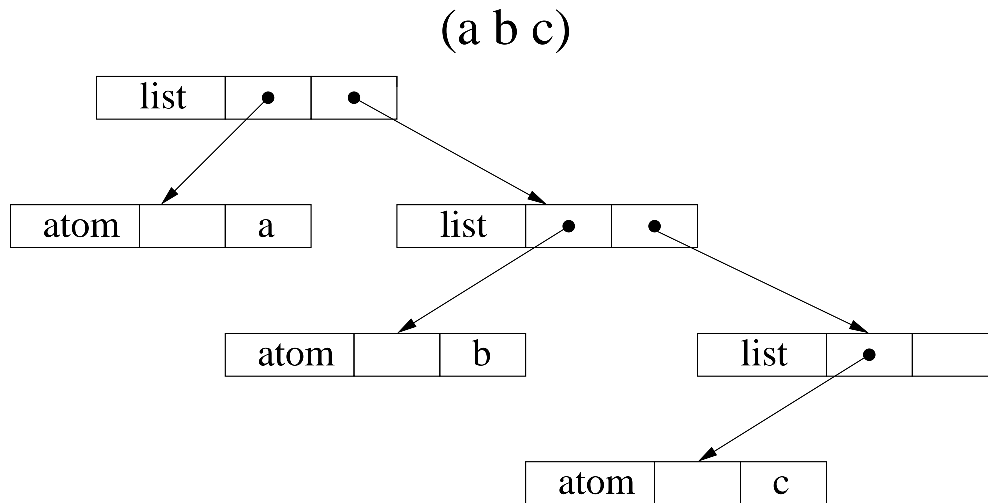
Esempio LISP:

```
(cons '(a b) '(d e)) = ((a b) d e)
```

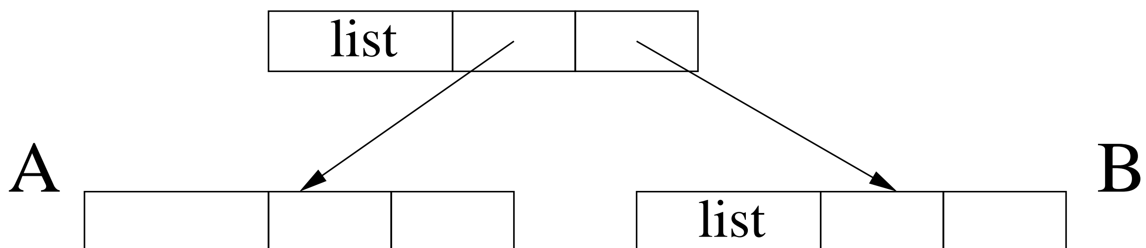
La lista risultante non è omogenea.

implementazione: rappresentazione in **memoria**





implementazione di **cons**



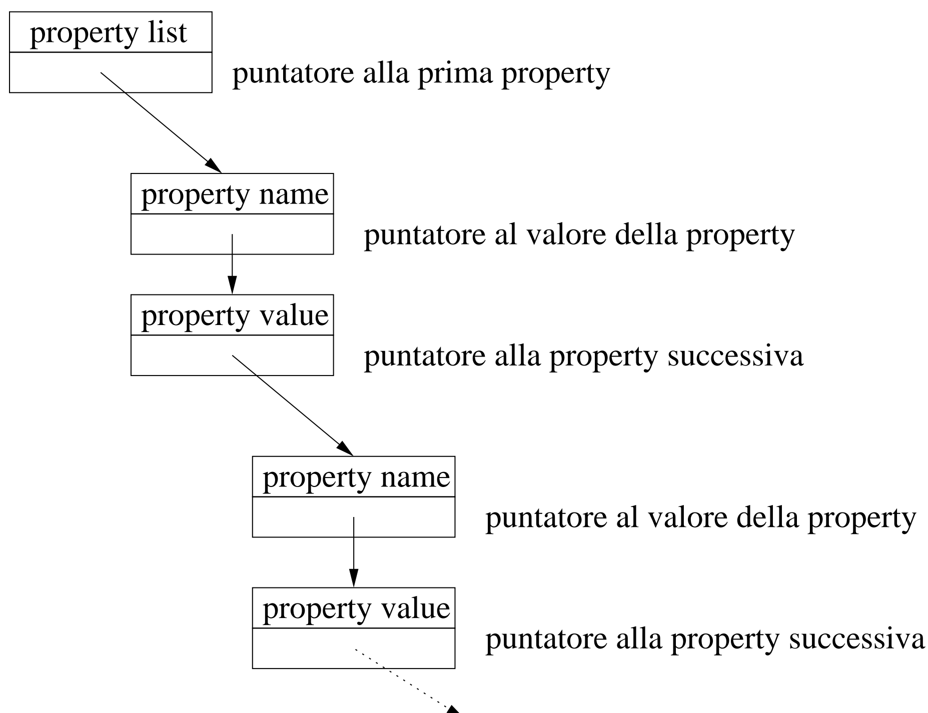
pile, alberi, grafi: sono tipi di dato

- o dell'utente
- o della macchina

sono "praticamente" assenti tra i tipi di dato forniti dai linguaggi

property list: record con un numero variabile (senza restrizioni) di componenti.

implementazione: rappresentazione in **memoria**



Stringhe di Caratteri

Pascal

```
nome:  packed array [1..10] of char;
```

C

```
char nome[10];
+ r-value(nome[9])='\0'
```

Lunghezza delle stringhe

- fissa
- limitata
- variabile

operazioni:

- concatenazione
  $\cdot$: `string` $\times$ `string` $\rightarrow$ `string`
- operazioni relazionali
- selezione sottostringhe

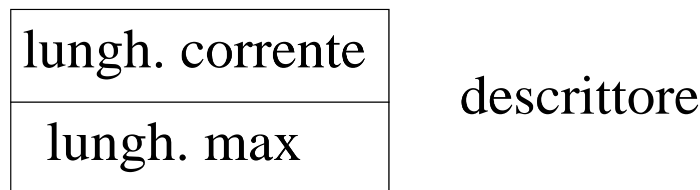
Esempio Fortran (sel. sttstrng)

`A=B(7:10)`

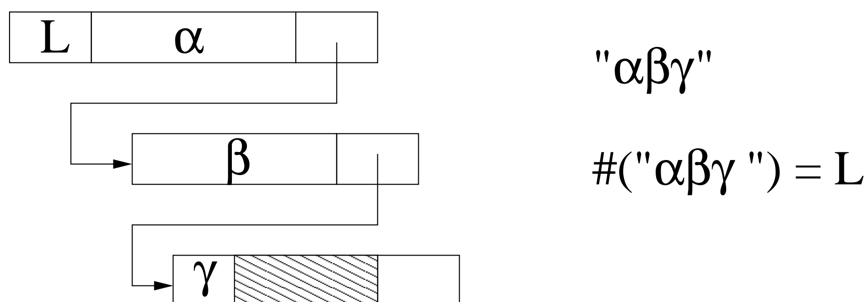
`A(6:10)=A(7:11)` $\Rightarrow$ possibile problema

implementazione: rappresentazione in **memoria**

- stringhe a lunghezza fissa: packed array
- stringhe a lunghezza limitata



- stringhe a lunghezza illimitata
  - a liste (con allocazioni fisse)



  - con terminatore (tecnica C)

Puntatori

Il linguaggi, piuttosto che includere tipi di dato di dimensioni variabili, permettono di costruire strutture (dinamiche) collegando (link) fra loro data object.

`p` è un puntatore $\Rightarrow$ `r-value(p)` = $\begin{cases} \texttt{l-value}(\alpha) \\ \text{NULL} \end{cases}$

```
void *p;
...
p=malloc(30);
```

operazioni: è usualmente presente un'operazione di

dereferenziamento (DR)
Se `p` puntatore

`r-value(DR(p))`

è il "valore" dell'oggetto con indirizzo

`r-value(p)`

operazioni in memoria

- allocazione dinamica
- deallocazione

(le esamineremo in seguito in dettaglio)

Esempio Pascal:

```
type ln = ↑cell;
        cell = record
                    inf: integer;
                    nx: ln;
                 end;
var p: ln;

new(p);
p↑.inf:=3;
p↑.nx:=q;
```

Esempio C:

```
struct cell;
    {int inf;
     struct cell *nx; }
struct cell *p;

p = (struct cell *) malloc(sizeof(struct cell));
(*p).inf=3;
(*p).nx=q;
```

implementazione

meccanismi HW (parte facile)

+

gestione HEAP: problema molto complesso, soprattutto il recupero della memoria deallocata (**garbage collection**) → argomento di ricerca

(le esamineremo in seguito in dettaglio)

Insiemi

È un data object contenente una collezione non ordinata di valori distinti.

operazioni

- appartenenza di un valore ad un insieme
- inserzione/eliminazione di singoli valori
- unione, intersezione, differenza di insiemi

implementazione

- stringa di bit (Pascal): ogni bit rappresenta la presenza (1) o l'assenza (0) di un singolo elemento. Limiti sulla cardinalità degli insiemi.
- codifica hash (più comune): elemento x rappresentato dalla stringa di bit B_x viene inserito nell'insieme S, rappresentato dal blocco di memoria M_S, nel seguente modo:

codifica hash

Ipotesi: numero di B_x distinte $>>$ locazioni in M_S

- `hash_function(`B_x`)` $\rightarrow$ `posiz.`I_x `per` x `in` M_S
- `se` I_x `libera` $\Rightarrow$ `inserzione`
- I_x `occupata ma da valore` $\neq x$ $\Rightarrow$ `collisione`
- `gestione collisione`
  - `reashing: modifica della funzione di hash (secondo un criterio fissato) fino a che si trova una locazione libera (o` x`)`
  - `scansione sequenziale: parto dalla locazione` I_x `e scandisco in avanti (o indietro) fino che non trovo una locazione libera (o` x`)`
  - `lista: in` I_x `non metto valori, ma un puntatore ad una lista di valori che va scandita linearmente per trovare` x `(o inserirlo in fondo)`

Tipi di Dato Interpretabili

Normalmente nei LdP:
`programmi sorgente` $\neq$ `dati che essi manipolano`

Ciò non è sempre vero:
in LISP, ML . . . i programmi sono “dati”

Esempio ML:
`[fn x` $\Rightarrow$ `3, fn y` $\Rightarrow$ `y+1]`

File e I/O

I file sono una struttura dati con due particolari proprietà:

- stanno in memoria di massa (molto grandi)
- tempo di vita superiore a quello dei programmi che li creano

Tipi di file:

- **sequenziali** (ascii/binari): file position pointer che specifica una posizione: (i) prima della prima componente; o (ii) tra due componenti; o (iii) dopo l'ultima componente
- ad **accesso diretto** (tramite chiave → intero o altro identificatore)
- **indexed sequential**: accesso diretto + scansione seq. a partire dalla componente selezionata

operazioni

- apertura/chiusura/creazione
- lettura/scrittura
- test EOF

implementazione: sulla macchina-sistema-operativo tramite chiamate di sistema

Esempio Pascal:

```
filen:  file of recordtype;
```

Esempio C: i file sequenziali non sono strutturati

```
FILE *p;
...
p=fopen(‘‘/usr/a’’, ‘‘r’’);
```