

L'astrazione

Metodi Avanzati di Programmazione

Laurea Triennale in Informatica

Università degli Studi di Bari Aldo Moro

Docente: Pierpaolo Basile

Credits

- Prof.ssa Annalisa Appice
- Prof. Michelangelo Ceci
- Prof. Donato Malerba

Astrazione nella
progettazione

L'astrazione

Astrarre: dal lat. *abstrahere* = 'trascinare via', dal verbo *trahere* = trascinare.

Cosa trascinare via? Un concetto, una idea, un principio da una realtà concreta.

In ambito scientifico, *astrarre significa cambiare la rappresentazione di un problema.*

Perché astrarre?

L'obiettivo del cambio di rappresentazione è quello di concentrarsi su aspetti rilevanti dimenticando gli elementi secondari

L'astrazione

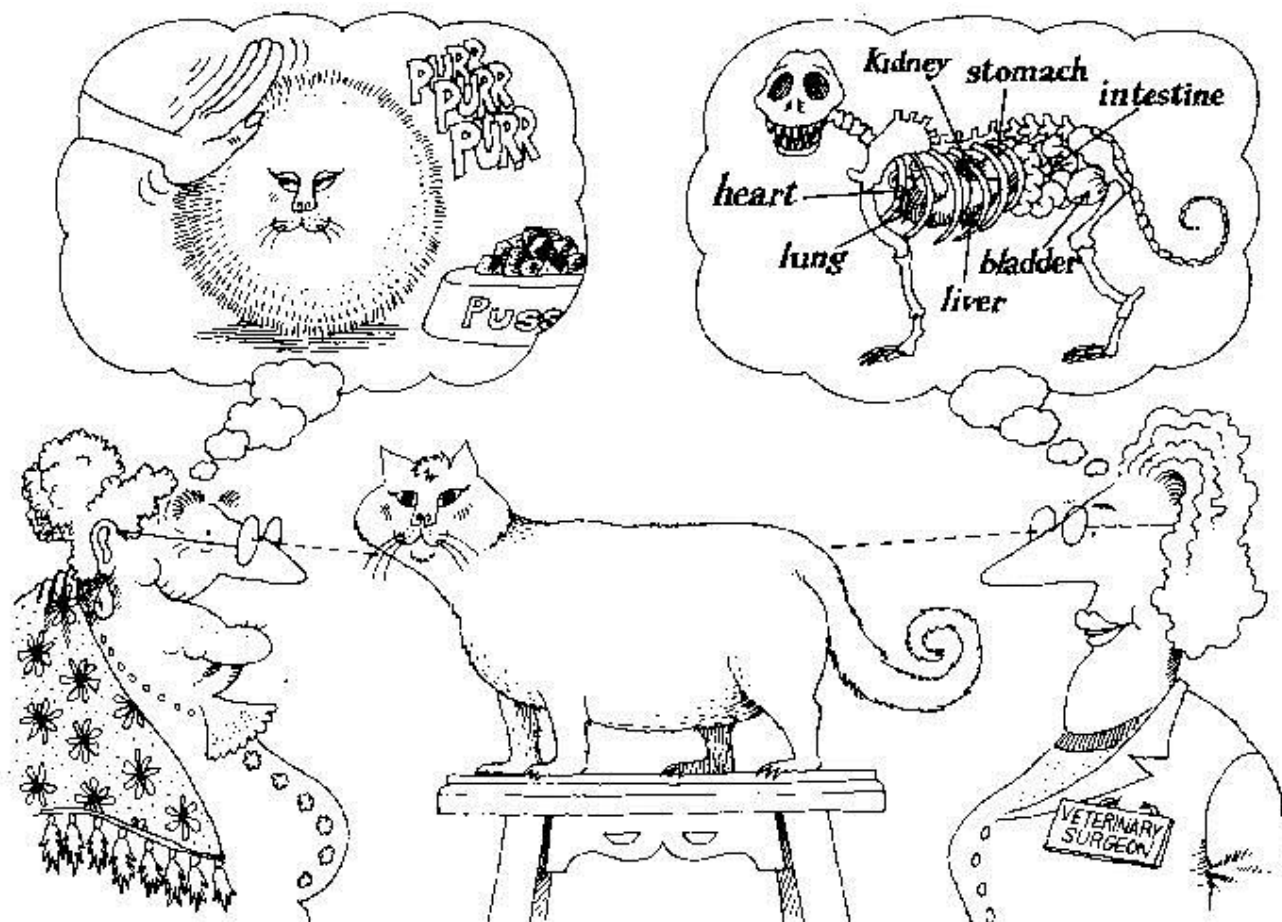
Non si tratta di **omettere** parti della rappresentazione di un problema, ma di **reformulare** lo stesso concentrando l'attenzione su idee generali piuttosto che su manifestazioni specifiche di quelle idee, tenendo conto della **prospettiva** di un osservatore.

Problema: trovare il percorso migliore per un commesso viaggiatore che deve visitare diverse città.

Se 'migliore' significa 'più breve', possiamo trascurare fattori come presenza di traffico e qualità della viabilità, e riformulare il problema come ricerca di minimo percorso in un grafo.



L'astrazione



L'astrazione si focalizza sulle caratteristiche essenziali di un oggetto, rispetto alla **prospettiva** di colui che osserva.

Astrazione: processo o entità

Il termine astrazione sotto-intende

- **Un processo**: l'estrazione delle informazioni essenziali e rilevanti per un particolare scopo, ignorando il resto dell'informazione
- **Una entità**: una descrizione semplificata di un sistema che enfatizza alcuni dei dettagli o proprietà trascurandone altri

Entrambe le viste sono valide e di fatto necessarie.

L'astrazione nell'analisi dei sistemi

Nell'**analisi dei sistemi** un'astrazione è una descrizione semplificata del sistema, che enfatizza alcuni dettagli o proprietà essenziali del sistema mentre ne ignora altri estranei.

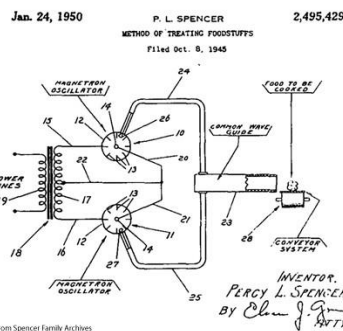
Esempio: controllo del traffico aereo

- Essenziale**: posizione del velivolo, velocità, etc.
- Irrilevante**: colore, nomi dei passeggeri, etc.

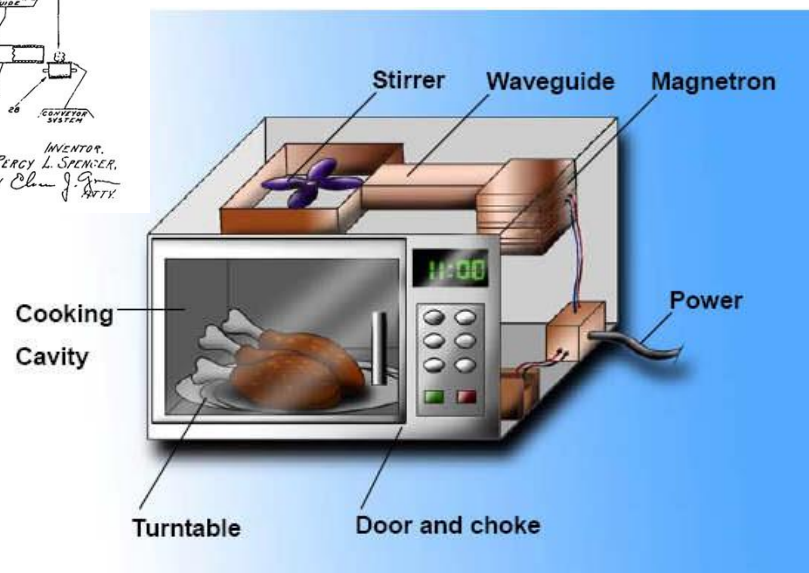
L'astrazione nell'analisi dei sistemi

Nel quotidiano il principio di astrazione è costantemente applicato ogni qualvolta utilizziamo uno strumento senza per questo sapere come è realizzato.

Esempio: utilizzo di un forno a microonde senza conoscerne la realizzazione.



brevetto



Astrazione e software

- Similmente, nella **programmazione** l'astrazione allude alla distinzione che si fa tra:

- cosa (*what*) fa un pezzo di codice
- come (*how*) esso è implementato

Per l'utente l'essenziale è cosa fa il codice mentre non è interessato ai dettagli della implementazione.

Astrazione e software

I sistemi software diventano sempre più complessi.

Per **padroneggiare la complessità** è necessario concentrarsi solo sui pochi aspetti che più interessano in un certo contesto ed ignorare i restanti.

L'**astrazione** permette ai progettisti di sistemi software di risolvere problemi complessi in modo **organizzato e gestibile**.

Astrazione funzionale

L'astrazione funzionale si riferisce alla **progettazione del software**, e in particolare alla possibilità di **specificare un modulo software che trasforma** dei dati di input in dati di output nascondendo i dettagli algoritmici della trasformazione.

- In sintesi:
 - Il modulo software deve trasformare un input in un output, cioè deve calcolare una funzione
 - I dettagli della trasformazione, cioè del calcolo, non sono visibili al consumatore (fruitore) del modulo.
 - Il consumatore conosce solo le corrette convenzioni di chiamata (**specifica sintattica**) e 'cosa' fa il modulo (**specifica semantica**).
 - Il consumatore deve fidarsi del risultato.

Astrazione funzionale

Esempio: modulo che realizza un operatore per il calcolo del fattoriale.

La *specificata sintattica* indica il nome del modulo (e.g., **fatt**), il tipo di dato in input (un intero) e il tipo di risultato (un intero), in modo da permettere la corretta chiamata del modulo. `fatt(intero) → intero`

La *specificata semantica* indica la trasformazione operata, cioè la funzione calcolata:

$$fatt(n) = \begin{cases} n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1 & n \geq 1 \\ 1 & n = 0 \end{cases}$$

Come la trasformazione è calcolata o **realizzata** (e.g., iterativamente o ricorsivamente) non è noto al fruitore del modulo.

Astrazione funzionale: specifiche semantiche

Problema: come specificare la semantica del modulo?

Un modo è quello di esprimere, mediante due **predicati**, la relazione che lega i dati di ingresso ai dati di uscita:

se il primo predicato (detto **precondizione**) è vero sui dati di ingresso e se il programma termina su quei dati, allora il secondo (detto **postcondizione**) è vero sui dati di uscita.

Queste specifiche semantiche sono dette **assiomatiche**.

Nell'esempio del fattoriale:

Precondizione: $n \in \mathbb{N}$

Postcondizione: $fatt(n) = \begin{cases} n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1 & n \geq 1 \\ 1 & n = 0 \end{cases}$

Astrazione funzionale: specifiche semantiche

Esercizio:

Un modulo ha in input un intero x e un vettore A di n interi e restituisce un intero p e un vettore B di n interi.

Siano date le seguenti:

Precondizione: $\{ n > 0 \wedge \forall i \in [1, n] A[i] \in \mathbb{Z} \}$

Postcondizione: $\{ \forall i \in [1, n] \exists j \in [1, n] B[i] = A[j] \wedge$
 $\forall i \in [1, p-1] B[i] \leq x \wedge \forall i \in [p+1, n] B[i] > x \}$

Riconoscete la funzione di questo modulo?

Stepwise refinement

L'astrazione funzionale si è affermata pienamente solo nei primi anni '70, quando emerse una metodologia che mirava a costruire i programmi progredendo dal generale al particolare (“**stepwise refinement**” o “**top-down programming**”).

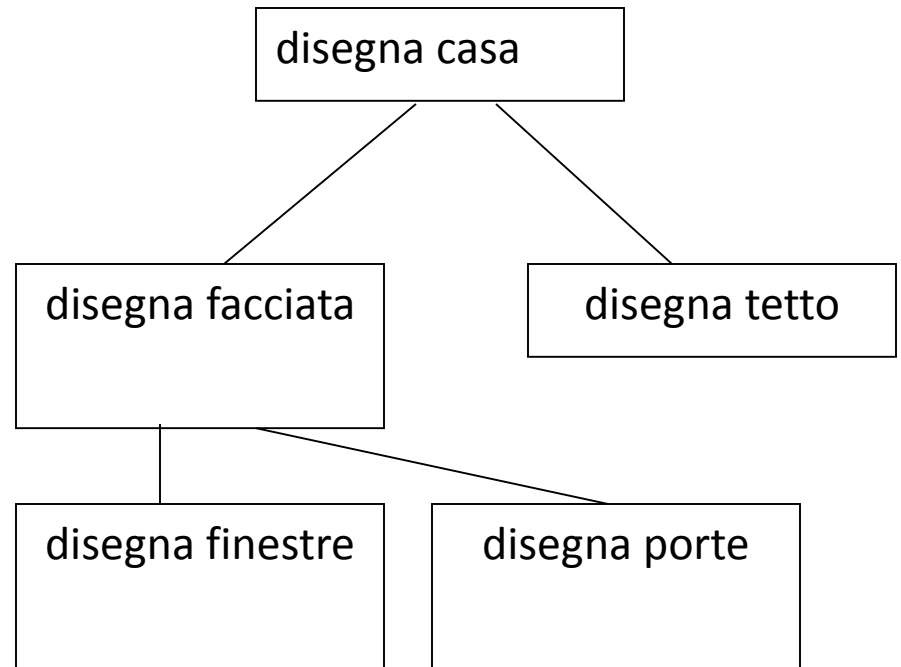
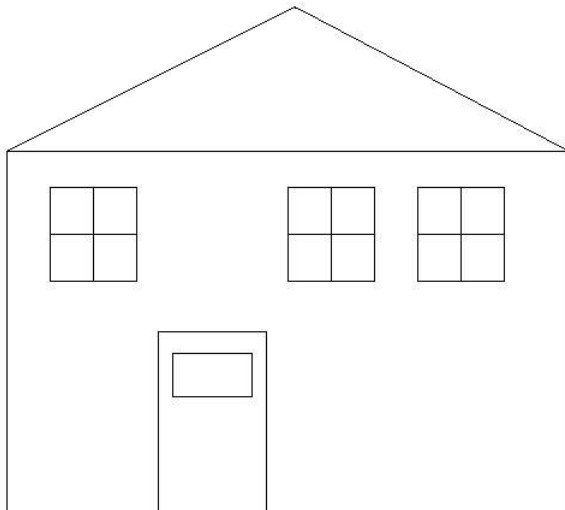
Secondo questa metodologia, per risolvere un problema (o affrontare un compito) P si procede come segue:

Stepwise refinement

1. Decomponi il compito P in sottocompiti P_1, P_2, \dots, P_n
2. Ipotizza di disporre di moduli M_1, M_2, \dots, M_n che effettuano le trasformazioni richieste rispettivamente da P_1, P_2, \dots, P_n
3. Componi un modulo M che assolve al compito P usando i moduli M_1, M_2, \dots, M_n
4. Applica ricorsivamente la metodologia ai sottocompiti P_1, P_2, \dots, P_n al fine di definire la realizzazione di M_1, M_2, \dots, M_n fino a quando non si ottengono sottocompiti considerati elementari (o non ulteriormente decomponibili).

Stepwise refinement

Tipicamente la dipendenza fra moduli è rappresentata da un albero.



Stepwise refinement

Secondo la metodologia di stepwise refinement, il programmatore è libero di assumere l'esistenza di qualsiasi modulo (detto *stub*, lett. *matrice* di qualcosa, e.g., assegno) che si può applicare al particolare sottocompito e di cui fornisce una specifica, salvo dover poi specificare come quel modulo va realizzato.

L'astrazione funzionale è di supporto alla metodologia dello stepwise refinement.

Limiti dell'astrazione funzionale

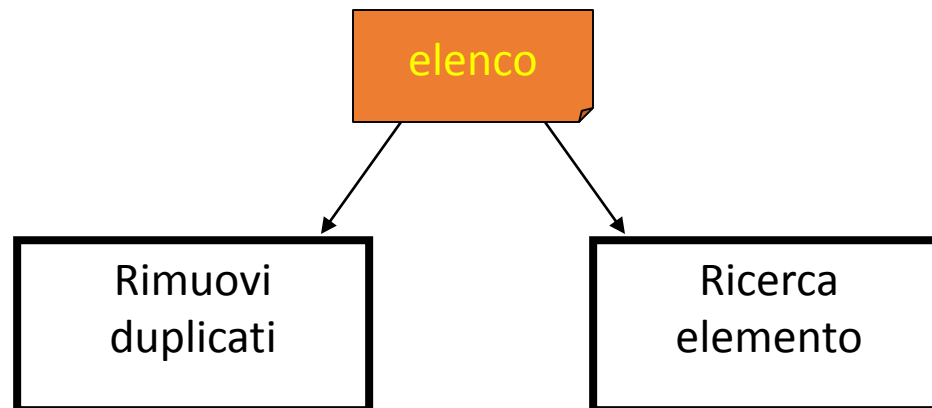
1. I dettagli relativi alla rappresentazione dei dati di input e output devono essere conosciuti da chi poi andrà a realizzare il modulo.

Esempio: un modulo che rimuove i duplicati in un elenco deve sapere se questo è realizzato con un array, un file, etc.

2. La rappresentazione è solitamente condivisa fra diversi moduli, per cui i cambiamenti apportati alla rappresentazione dei dati in input/output a un modulo si possono ripercuotere su molti altri moduli.

Limiti dell'astrazione funzionale

Esempio: due moduli operano su uno stesso elenco, uno per rimuovere duplicati e l'altro per ricercare un elemento. Per migliorare l'efficienza dell'ultimo sarebbe opportuno ricorrere a una rappresentazione dell'elenco con tabelle hash. Tuttavia ciò comporta un cambiamento nel primo modulo, che dovrà necessariamente operare la trasformazione (rimozione dei duplicati) in modo differente.



Limiti dell'astrazione funzionale

L'astrazione funzionale non permette di sviluppare soluzioni **invarianti ai cambiamenti nei dati** (sono invarianti solo ai cambiamenti nei processi di trasformazione che operano).

Questo rende difficoltosa la manutenzione delle soluzioni progettate.

È inappropriata per lo sviluppo di soluzioni a problemi complessi.

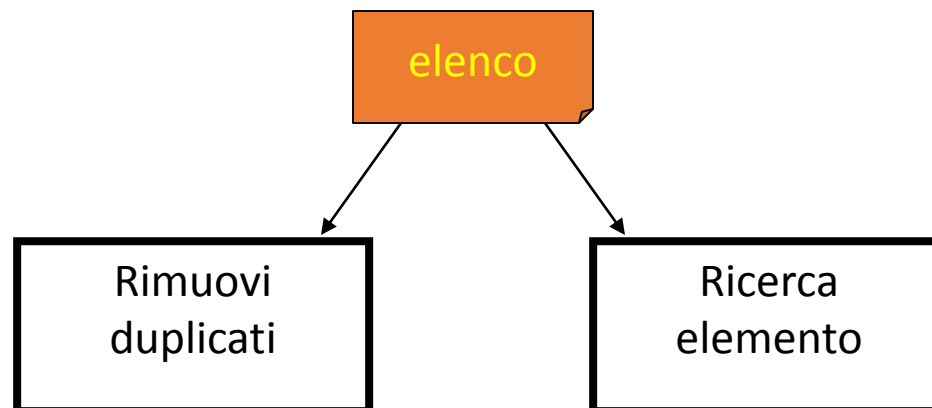
Astrazione dati

Alla base dell'**astrazione dati** c'è il principio che *non si può accedere direttamente alla rappresentazione di un dato, qualunque esso sia, ma solo attraverso un insieme di operazioni considerate lecite*
(**principio dell'astrazione dati**).

VANTAGGIO: un cambiamento nella rappresentazione del dato si ripercuoterà solo sulle operazioni lecite, che potrebbero subire delle modifiche, mentre non inficerà il codice che utilizza il dato astratto.

Astrazione dati

Esempio: Se i moduli 'Rimuovi duplicati' e 'Ricerca elemento' accedono all'elenco attraverso un insieme di operazioni lecite (e.g., 'dammi il prossimo elemento', 'non ci sono più elementi', ecc.) il cambiamento della rappresentazione dell'elenco richiederà una riformulazione delle operazioni lecite, ma non influenzerà i due moduli.



Information Hiding

In generale un principio di astrazione suggerisce di **occultare l'informazione** (*information hiding*) sulla rappresentazione del dato

- sia perché non necessaria al fruitore dell'entità astratta
- sia perché la sua rivelazione creerebbe delle inutili dipendenze che comprometterebbero l'invarianza ai cambiamenti

Information Hiding

Il principio dell'astrazione funzionale suggerisce di occultare i dettagli del **processo di trasformazione** (“come” esso è operato).

Il principio dell'astrazione dati identifica nella **rappresentazione del dato** l'informazione da nascondere.

In entrambi i casi non si dice COME farlo.

Questo sarà chiarito quando approfondiremo il tema dell'astrazione nella programmazione.

Incapsulamento

L'**incapsulamento** (**encapsulation**) è una tecnica di progettazione consistente nell'impacchettare (o "racchiudere in capsule") una collezione di entità, creandone una barriera concettuale.

Come l'astrazione, l'incapsulamento sottointende

- *Un processo*: l'impacchettamento
- *Una entità*: il «pacchetto» ottenuto

Incapsulamento

Esempi:

- una procedura impacchetta diversi comandi
- una libreria incapsula diverse funzioni
- un oggetto incapsula un dato e un insieme di operazioni sul dato

Incapsulamento

L'incapsulamento NON dice come devono essere le “pareti” del pacchetto (o capsula), che potranno essere:

- **Trasparenti**: permettendo di vedere **tutto** ciò che è stato impacchettato
- **Traslucide**: permettendo di **vedere** in modo **parziale** il contenuto
- **Opache**: **nascondendo tutto** il contenuto del pacchetto

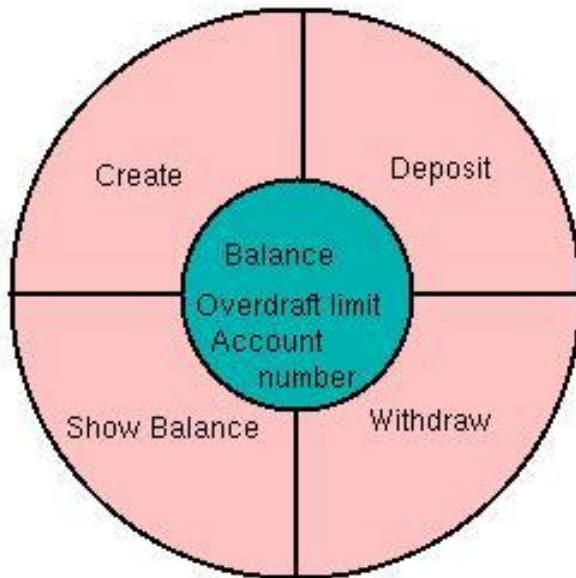
Astrazione dati & Incapsulamento

La combinazione del principio dell'astrazione dati con la tecnica dell'incapsulamento suggerisce che:

1. La rappresentazione del dato va nascosta
2. L'accesso al dato deve passare solo attraverso operazioni lecite
3. Le operazioni lecite, che ovviamente devono avere accesso alla informazione sulla rappresentazione del dato, vanno impacchettate con la rappresentazione del dato stesso

Astrazione dati & Incapsulamento

Esempio: il dato “conto corrente” ha una sua rappresentazione interna che permette di memorizzare il saldo (*balance*), il limite fido (*overcraft limit*) e il numero di conto (*account number*). La rappresentazione interna dei tre dati è nascosta. L'accesso alla rappresentazione passa per tre operazioni lecite:

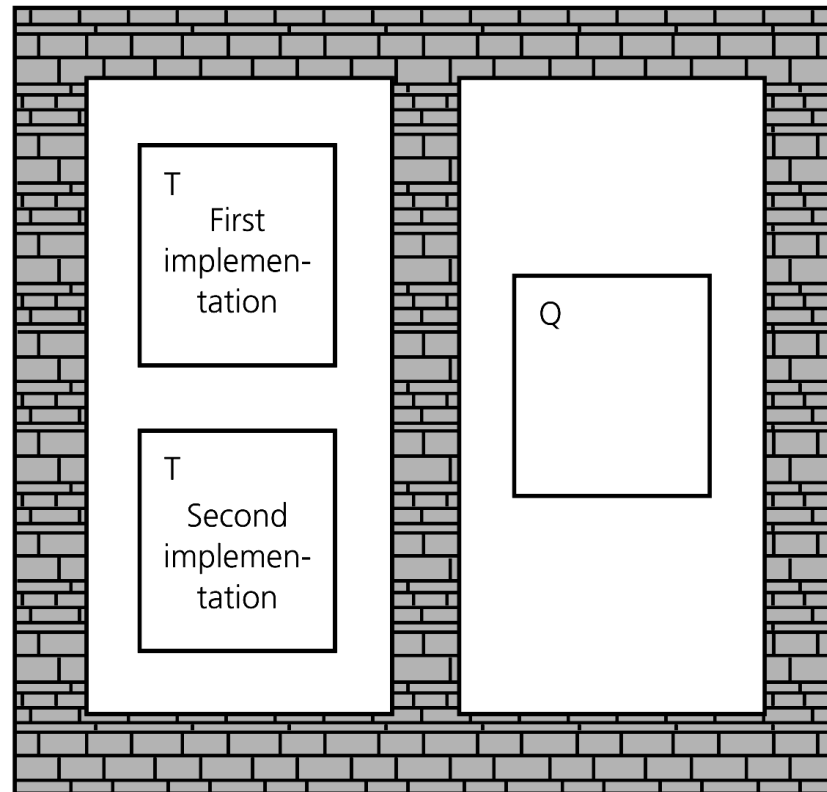


- Creazione conto
- Deposito
- Prelievo
- Stampa saldo

Rappresentazione e operazioni lecite sono impacchettate in un modulo.

Astrazione dati & Incapsulamento

T e Q sono isolati: l'implementazione di T non influenza Q



Astrazione dati vs. Astrazione funzionale

L'astrazione dati ricalca ed **estende** quella funzionale.

Attualmente la possibilità di effettuare astrazioni di dati è considerata importante almeno quanto quella di definire nuovi operatori con astrazioni funzionali. L'esperienza ha infatti dimostrato che la scelta delle strutture di dati è il primo passo sostanziale per un buon risultato dell'attività di programmazione.

Astrazione dati vs. Astrazione funzionale

L'astrazione funzionale stimola gli sforzi per evidenziare operazioni ricorrenti o comunque ben caratterizzate all'interno della soluzione di un dato problema.

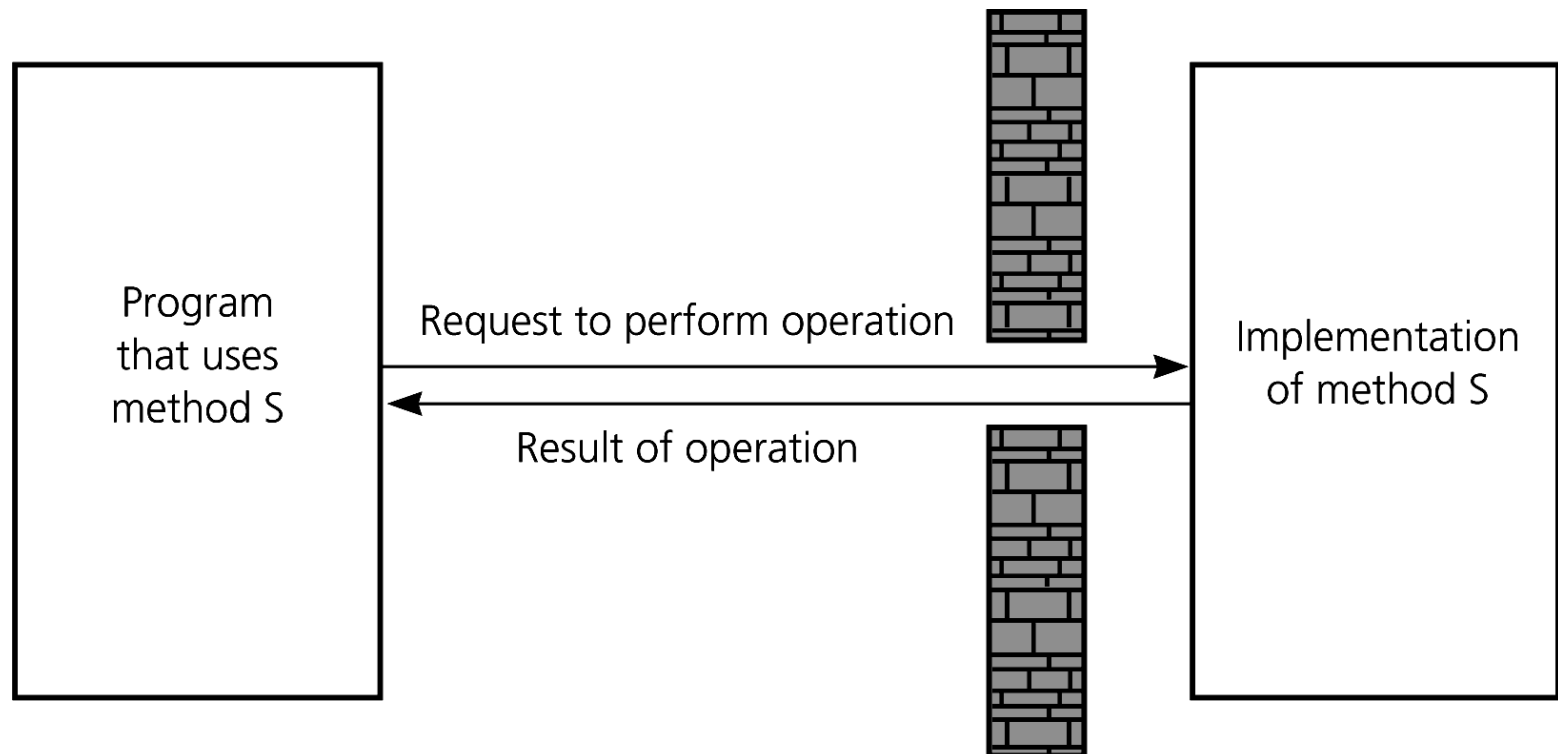
L'astrazione di dati stimola in più gli sforzi per individuare le organizzazione dei dati più consone alla soluzione del problema.

Da una progettazione *function centered* a una *data centered*.

Astrazione dati & Incapsulamento

Ovviamente, l'isolamento dei moduli non può essere totale.

- La **specifica**, o **contratto**, descrive come si può interagire con un dato astratto.



Astrazione: i punti di vista

In generale, le astrazioni supportano la separazione dei diversi interessi di

- **Utenti**: interessati a cosa si astrae (*what*)
- **Implementatori**: interessati a come (*how*) si realizza

Per questa ragione una definizione di astrazione ha sempre due componenti:

- **Specifica**
- **Realizzazione**

Per descrivere una specifica occorre ricorrere a dei *linguaggi di specifica*, che sono diversi dai linguaggi usati per descrivere le realizzazioni delle astrazioni.

Specifica sintattica e semantica

La specifica potrà poi essere:

- **Sintattica**: stabilisce quali identificatori sono associati all'astrazione
- **Semantica**: definisce il risultato della computazione inclusa nell'astrazione.

Parametrizzazione di un'astrazione

L'efficacia di un'astrazione può essere migliorata mediante l'uso di **parametri** per la comunicazione con l'ambiente esterno.

Quando un'astrazione è chiamata, ciascun **parametro formale** verrà associato in qualche modo al corrispondente **argomento**.

I meccanismi utilizzati per realizzare queste associazioni sono fondamentalmente due:

- **Meccanismi di copia**: copiano il valore da passare
- **Meccanismi definizionali**: legano direttamente il parametro formale alla definizione dell'argomento passato

Astrazione dati

Anche un'astrazione dati, come una qualunque astrazione, è costituita da una *specificata* e una *realizzazione*:

- La *specificata* consente di descrivere un nuovo dato e gli operatori che ad esso sono applicabili
- La *realizzazione* stabilisce come il nuovo dato e i nuovi operatori vengono ricondotti ai dati e agli operatori già disponibili

Chi intende usare i nuovi dati con i nuovi operatori nella scrittura dei suoi programmi sarà tenuto a conoscere la specifica dell'astrazione dei dati, ma potrà astrarre dalle tecniche utilizzate per la realizzazione.

Astrazione dati

I linguaggi di specifica per astrazione dati più noti sono due:

- Il linguaggio logico-matematico usato nelle asserzioni → **specifiche assiomatiche**
- Il linguaggio dell'algebra usato nelle equazioni definite fra gli operatori specificati nel dato astratto → **specifiche algebriche.**

Astrazione dati: le specifiche assiomatiche

Un linguaggio formale per la specifica di un tipo astratto di dato è costituito dalla notazione logico-matematica delle **asserzioni**. In questo caso si parla di specifica assiomatica.

Una specifica assiomatica consta di:

- a) Una specifica **sintattica** (detta **segnatura**, *signature*) che fornisce:
 - 1) l'elenco dei NOMI dei domini e delle operazioni specifiche del tipo;
 - 2) i DOMINI di partenza e di arrivo per ogni nome di operatore;

Astrazione dati: le specifiche assiomatiche

b) Una specifica **semantica** che associa:

- 1) un INSIEME ad ogni nome di tipo introdotto nella specifica sintattica;
- 2) una FUNZIONE ad ogni nome di operatore, esplicitando le seguenti condizioni sui domini di partenza e di arrivo:
 - i) *precondizione*, che definisce quando l'operatore è applicabile
 - ii) *postcondizione*, che stabilisce la relazione tra argomenti e risultato

Il dato astratto *Vettore*

Un esempio elementare di specifica assiomatica è quella di un vettore

a) Specifica sintattica

Tipi: vettore, intero, tipoelem

Operatori:

CREAVETTORE() → vettore

LEGGIVETTORE(vettore, intero) → tipoelem

SCRIVIVETTORE(vettore, intero, tipoelem) → vettore

Il dato astratto *Vettore*

b) Specifica semantica

Tipi:

intero: l'insieme dei numeri interi

vettore: l'insieme delle sequenze di n elementi di tipo *tipoelem*

Operatori:

CREAVETTORE = v

Pre: *non ci sono precondizioni, o più precisamente, il predicato che definisce la precondizione di applicabilità di CREAVETTORE è il valore VERO*

Post: $\forall i \in \{0, 1, 2, \dots, n-1\}$, l' i -esimo elemento del vettore, $v(i)$, è uguale ad un prefissato elemento di tipo *tipoelem*

LEGGIVETTORE(v, i) = e

Pre: $0 \leq i \leq n-1$

Post: $e = v(i)$

SCRIVIVETTORE(v, i, e) = v'

Pre: $0 \leq i \leq n-1$

Post: $\forall j \in \{0, 1, \dots, n-1\}, j \neq i, v'(j) = v(j), v'(i) = e$

Il dato astratto *Vettore*

c) Realizzazione

In Java il vettore (array) è un tipo di dato concreto. La corrispondenza tra la specifica appena introdotta e quella del Java è la seguente:

CREAVETTORE \Leftrightarrow tipoelem $v[n]$

LEGGIVETTORE(v, i) \Leftrightarrow $v[i]$

SCRIVIVETTORE(v, i, e) \Leftrightarrow $v[i] = e$

dove i può essere anche una espressione di tipo intero.

Il dato astratto *Pila*

Una pila (*stack*) è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi solo da un estremo della sequenza (la “testa”).

Può essere vista come un caso speciale di lista in cui l'ultimo elemento inserito è il primo ad essere rimosso (LIFO, ***last in, first out***) e non è possibile accedere ad alcun elemento che non sia quello in testa.

Il dato astratto *Pila*

Una specifica assiomatica di una pila

Specifica sintattica

Tipi: pila, booleano, tipoelem

Operatori:

CREAPILA() \rightarrow pila

PILAVUOTA(pila) \rightarrow booleano

LEGGIPILA(pila) \rightarrow tipoelem

FUORIPILA(pila) \rightarrow pila

INPILA(tipoelem, pila) \rightarrow pila

Il tipo astratto *Pila*

Specifica semantica

Tipi:

pila = insieme delle sequenze di elementi di tipo
tipoelem

booleano = insieme dei valori di verità {vero,falso}

Operatori:

CREAPILA ()=P'

Post: $P' = \wedge$, la sequenza vuota

PILAVUOTA(P)=b

Post: $b = \text{vero}$, se $P = \wedge$; $b = \text{falso}$, altrimenti

Il tipo astratto *Pila*

LEGGIPILA(P)=a

Pre: $P = a_1, a_2, \dots, a_n$ e $n \geq 1$

Post: $a = a_1$

FUORIPILA(P)=P'

Pre: $P = a_1, a_2, \dots, a_n$ e $n \geq 1$

Post: $P' = a_2, \dots, a_n$ se $n > 1$; $P' = \wedge$ se $n = 1$

INPILA(a,P)=P'

Pre: $P = a_1, a_2, \dots, a_n$ e $n \geq 0$

Post: $P' = a, a_1, a_2, \dots, a_n$ se $n > 0$; $P' = a$ se $n = 0$

Il dato astratto *Coda*

Una coda è un dato astratto che consente di rappresentare una sequenza di elementi in cui è possibile aggiungere elementi ad un estremo (“il fondo”) e togliere elementi dall’altro estremo (“la testa”).

Tale disciplina di accesso è detta FIFO (**First In First Out**).

È adatta a rappresentare sequenze nelle quali l’elemento viene elaborato secondo l’ordine di arrivo (lista d’attesa, etc.)

Il dato astratto *Coda*

Una specifica assiomatica di una coda

Specifica sintattica

Tipi: coda, booleano, tipoelem

Operatori:

CREACODA() \rightarrow coda

CODAVUOTA(coda) \rightarrow booleano

LEGGICODA(coda) \rightarrow tipoelem

FUORICODA(coda) \rightarrow coda

INCODA(tipoelem, coda) \rightarrow coda

Il dato astratto *Coda*

Specifica semantica

Tipi:

coda = insieme delle sequenze $\langle a_1, a_2, \dots, a_n \rangle$, con $n > 0$, di elementi di tipo *tipoelem*. Λ denota la sequenza vuota.

booleano = insieme dei valori di verità {vero, falso}

Operatori:

CREACODA() $=q'$

Post: $q' = \Lambda$

CODAVUOTA(q) $=b$

Post: $b = \text{vero}$, se $q = \Lambda$; $b = \text{falso}$, altrimenti

LEGGICODA(q) $=a$

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $a = a_1$

Il dato astratto *Coda*

Specifica semantica (cont.)

FUORICODA(q)= q'

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $q' = \langle a_2, \dots, a_n \rangle$ se $n > 1$; $q' = \Lambda$ se $n = 1$

INCODA(a, Q)= Q'

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle$ oppure $q = \Lambda$

Post: $q' = \langle a_1, a_2, \dots, a_n, a \rangle$, se $q \neq \Lambda$;

$q' = \langle a \rangle$ se $q = \Lambda$

Esercizio

Si vuole progettare un dato astratto *Deque* (*double-ended queue* o *coda doppia*), una struttura lineare che permette di inserire, cancellare, esaminare elementi solo dalle due estremità.

Dare le specifiche assiomatiche (semantiche) di *deque*, tenuto conto delle seguenti specifiche sintattiche:

Esercizio

Tipi

deque, item, boolean

Operatori

new() → deque

addq(item, deque) → deque

bottom(deque) → item

leaveq(deque) → deque

push(item, deque) → deque

top(deque) → item

pop(deque) → deque

isnew(deque) → boolean

crea nuove code doppie

accoda un elemento

restituisce l'elemento di coda

estrae l'elemento di coda

impila un elemento

restituisce l'elemento di testa

estrae l'elemento di testa

stabilisce se una coda è vuota

Esercizio

Specifica semantica

Tipi:

deque = insieme delle sequenze $\langle a_1, a_2, \dots, a_n \rangle$, con $n > 0$, di elementi di tipo *item*. Λ denota la sequenza vuota.

boolean = insieme dei valori di verità {vero, falso}

Operatori:

new () = q'

Post: $q' = \Lambda$

Esercizio

Specifica semantica (cont.)

$\text{addq}(a,q)=q'$ // accoda un elemento

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle$ oppure $q = \Lambda$

Post: $q' = \langle a_1, a_2, \dots, a_n, a \rangle$, se $q \neq \Lambda$;
 $q' = \langle a \rangle$ se $q = \Lambda$

$\text{bottom}(q)=a$ // restituisce l'elemento di coda

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $a = a_n$

$\text{leaveq}(q)=q'$ //estrae l'elemento di coda

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $q = \langle a_1, a_2, \dots, a_{n-1} \rangle$ se $n > 1$; $q' = \Lambda$ se $n = 1$

Esercizio

Specifica semantica (cont.)

$\text{push}(a,q)=q'$ // impila un elemento

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle$ oppure $q = \Lambda$

Post: $q' = \langle a, a_1, a_2, \dots, a_n \rangle$, se $q \neq \Lambda$;
 $q' = \langle a \rangle$ se $q = \Lambda$

$\text{top}(q)=a$ // restituisce l'elemento di testa

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $a = a_1$

$\text{pop}(q)=q'$ // estrae l'elemento di testa

Pre: $q = \langle a_1, a_2, \dots, a_n \rangle \neq \Lambda$

Post: $q = \langle a_2, \dots, a_n \rangle$ se $n > 1$; $q' = \Lambda$ se $n = 1$

Esercizio

Specifica semantica (cont.)

isnew(q)=b // stabilisce se una coda doppia è vuota

Post: b=vero, se $q=\Lambda$; b=falso, altrimenti

Esercizio

Si vuole progettare il dato astratto *volo*.

Dare le specifiche assiomatiche (semantiche) di *volo*, tenuto conto delle seguenti specifiche sintattiche

Esercizio

Tipi

volo, passeggero, posto, integer, boolean

Operatori

creavolo() → volo

crea un nuovo volo

postolibero(posto, volo) → boolean
libero o meno

stabilisce se il posto è

check-in(posto, passeggero, volo) → volo
aggiunge un passeggero al volo assegnandogli il posto specificato, purché libero

libera(posto, volo) → volo

libera un posto sul volo

sostituisci(posto, passeggero, volo) → volo
assegna a un altro passeggero un posto già occupato

passeggero(posto, volo) → passeggero
che occupa un posto

restituisce il passeggero

passeggeri(volo) → integer

restituisce il numero di passeggeri con carta di imbarco, cioè che hanno fatto il check-in

Esercizio

Suggerimento:

posto è l'insieme K dei posti

passaggero è l'insieme P dei passeggeri.

volo è l'insieme di voli, dove ogni volo è a sua volta un insieme di coppie $(k, p) \in K \times P$.

integer è l'insieme Z dei numeri interi. 0 e 1 sono particolari valori dell'insieme Z .

boolean è l'insieme dei valori di verità $\{\text{true}, \text{false}\}$.

N.B.: Questa definizione è sufficientemente astratta e non dice nulla sull'insieme K . Infatti i posti su un aereo sono generalmente identificati da una coppia

Numero Lettera

Esercizio

Specifica semantica

Tipi:

volo = insieme di coppie $(k,p) \in K \times P$ dove K è l'insieme dei posti e P è l'insieme dei passeggeri. \emptyset denota l'insieme vuoto.

integer = insieme di numeri naturali

boolean = insieme dei valori di verità {vero, falso}

Operatori:

creavolo() $\rightarrow v'$ // crea un nuovo volo

Post: $v' = \emptyset$

Esercizio

Specifica semantica (cont.)

postolibero(k, v) = b // stabilisce se il posto è libero o meno

Pre: $v \subseteq K \times P, k \in K$

Post: $b = \text{vero}$ se $\neg(\exists p \in P (k, p) \in v)$, falso altrimenti

check-in(k, p, v) = v' // aggiunge un passeggero al volo assegnandogli il posto specificato, purché libero

Pre: $v \subseteq K \times P, k \in K, p \in P \neg(\exists q \in P (k, q) \in v)$,

Post: $v' = v \cup \{(k, p)\}$

Esercizio

Specifica semantica (cont.)

$\text{libera}(k, v) = v'$ // libera un posto sul volo

Pre: $v \subseteq K \times P$, $k \in K$, $\exists q \in P (k, q) \in v$

Post: $v' = v - \{(k, q)\}$

$\text{sostituisci}(k, p, v) = v'$ // assegna a un altro passeggero un posto già occupato

Pre: $v \subseteq K \times P$, $k \in K$, $p \in P$, $\exists q \in P (k, q) \in v$

Post: $v' = (v - \{(k, q)\}) \cup \{(k, p)\}$

Esercizio

Specifica semantica (cont.)

$\text{passaggero}(k, v)=p$ //restituisce il passeggero che occupa un posto

Pre: $v \subseteq K \times P, k \in K, \exists q \in P (k, q) \in v$

Post: $p=q$

$\text{passaggeri}(v)=n$ //restituisce il numero di passeggeri con carta di imbarco, cioè che hanno fatto il check-in

Pre: $v \subseteq K \times P$

Post: $n=|v|$

Limiti delle specifiche assiomatiche

Si noti che il metodo di specifica assiomatica è preciso nella definizione della specifica sintattica, mentre **è piuttosto informale per gli altri aspetti** (talvolta si ricorre al linguaggio naturale per semplicità). Pertanto esso non consente di caratterizzare precisamente un tipo astratto.

In particolare non consente di definire i valori che possono essere generati mediante l'applicazione di operatori, e non consente di stabilire quando l'applicazione di diverse sequenze di operatori porta al medesimo valore.

Questo problema è superato dalle specifiche algebriche.

Dato Astratto: specifiche algebriche

Le specifiche algebriche si basano sull'algebra, piuttosto che sulla logica. Essenzialmente definiscono un dato astratto come un'algebra eterogenea, ovvero come una collezione di diversi insiemi su cui sono definite diverse operazioni.

Le algebre tradizionali sono omogenee. Un'algebra omogenea consiste in un unico insieme e diverse operazioni.

Esempio: \mathbb{Z} con le operazioni di addizione e moltiplicazione è un'algebra omogenea.

Dato Astratto: specifiche algebriche

Esempio: dato un alfabeto Σ , indichiamo con Σ^* l'insieme di tutte le stringhe, incluso quella vuota, costruite con i simboli di Σ .

Σ^* con le operazioni di concatenazione e calcolo della lunghezza non sono un'algebra omogenea, visto che il codominio dell'operazione di calcolo della lunghezza è \mathbb{N} e non Σ^* .

Quindi quest'algebra consiste di due insiemi, stringhe e interi, su cui sono definite le operazioni di concatenamento e lunghezza delle stringhe.

Dato Astratto: specifiche algebriche

Una specifica algebrica consiste di tre parti:

- 1. Sintattica:** elenca i nomi del tipo, le sue operazioni e il tipo degli argomenti delle operazioni. Se un'operazione è una funzione allora è specificato il codominio (range) della funzione.
- 2. Semantica:** consiste di un insieme di equazioni algebriche che descrivono in modo indipendente dalla rappresentazione le proprietà delle operazioni.
- 3. Di restrizione:** stabilisce varie condizioni che devono essere soddisfatte o prima che siano applicate le operazioni o dopo che esse siano state completate.

N.B.: alcuni autori inglobano le specifiche di restrizione in quelle semantiche.

Dato Astratto: specifiche algebriche

Uno degli aspetti più interessanti delle specifiche algebriche è la semplicità del linguaggio di specifica rispetto ai linguaggi di programmazione procedurale.

Infatti, il linguaggio di specifica consiste di solo cinque primitive:

1. composizione funzionale
2. relazione di eguaglianza
3. 4. due costanti, true e false
4. un numero illimitato di variabili libere

Dato Astratto: specifiche algebriche

La funzione *if then else* può essere facilmente descritta dalle seguenti equazioni:

$$\textit{if then else} (\textit{true}, q, r) = q$$

$$\textit{if then else} (\textit{false}, q, r) = r$$

Questa funzione è così importante che si assume già data come operatore infisso:

$$\textit{if } p \textit{ then } q \textit{ else } r$$

Inoltre si assume che sono predefiniti i valori interi e booleani.

Specifica algebrica di *Pila*

Specifica sintattica

sorts: stack, item, boolean

operations:

newstack() → stack

push(stack, item) → stack

pop(stack) → stack

top(stack) → item

isnew(stack) → boolean

Ogni insieme è detto un **sort** (letteralmente “tipo”) dell'algebra eterogenea.

I *sort* **item** e **boolean** sono **ausiliari** alla definizione di stack.

Specifica algebrica di *Pila*

Specifica semantica

declare **stk**: stack, **i**: item

pop(push(stk, i)) = stk

top(push(stk, i)) = i

isnew(newstack) = true

isnew(push(stk, i)) = false

Nella specifica semantica
occorre **dichiarare** i
parametri su cui lavorano
gli operatori.

Specifica algebrica di *Pila*

Specifica di restrizione

restrictions

pop(newstack) = error

top(newstack) = error

dove 'error' è un elemento speciale indefinito.

Specifica algebrica di *Pila*

Ovviamente avremmo potuto scrivere molte altre equazioni, come:

$$isnew(pop(push(newstack, i))) = true$$

che evidenzia come il predicato *isnew* sia vero anche per quegli stack ottenuti inserendo un generico elemento *i* in un **nuovo** stack e poi rimuovendolo.

Tuttavia questa equazione è **ridondante**, poiché è ricavabile dalle altre scritte in precedenza. Infatti, grazie alla prima e terza equazione possiamo scrivere:

$$isnew(pop(push(newstack, i))) = isnew(newstack) = true$$

Specifica algebrica di *Pila*

Nelle specifiche semantiche è importante indicare l'insieme minimale di equazioni (dette **assiomi**) a partire dalle quali possiamo derivare tutte le altre.

Le specifiche semantiche si diranno **incomplete** se non permetteranno di derivare tutte le verità (equazioni) desiderate dell'algebra specificata.

Le specifiche semantiche si diranno **inconsistenti** (o contraddittorie) se permetteranno di derivare delle equazioni indesiderate, cioè considerate false.

Le specifiche semantiche si diranno **ridondanti** se alcune delle equazioni sono ricavabili dalle altre.

Costruttori e Osservazioni

Scrivere delle specifiche semantiche complete, consistenti e non ridondanti può non essere un compito semplice.

Per questo conviene introdurre una **metodologia**, che si basa sulla distinzione degli operatori di un dato astratto in:

- **Costruttori**, che creano o istanziano il dato astratto
- **Osservazioni**, che ritrovano informazioni sul dato astratto

Il comportamento di una astrazione dati può essere specificata riportando il valore di **ciascuna osservazione applicata a ciascun costruttore**.

Questa informazione è organizzata in modo naturale in una matrice, con i costruttori lungo una dimensione e le osservazioni lungo l'altra.

Costruttori e Osservazioni

<i>osservazioni</i>	<i>Costruttore di stk'</i>	
	<i>newstack</i>	<i>push(stk, i)</i>
<i>pop(stk')</i>	error	stk
<i>top(stk')</i>	error	i
<i>isnew(stk')</i>	true	false

Osservazioni binarie

Tutte le osservazioni viste finora sono unarie, nel senso che esse osservano un singolo valore del dato astratto.

Spesso è necessario disporre di osservazioni più complesse.

Per confrontare due valori è necessario osservare due istanze del dato astratto.

Questo complica la specifica, perché il valore dell'osservazione dev'essere definito per tutte le **combinazioni di costruttori** possibili per i valori astratti che si devono confrontare.

Osservazione binaria: $\text{equal}(l, m)$

Esempio: predicato $\text{equal}(l, m)$ che è vero solo se le due pile contengono gli stessi elementi nello stesso ordine.

<i>Costruttore di m</i>	<i>Costruttore di l</i>	
	<i>newstack</i>	<i>push(stk, i)</i>
<i>newstack</i>	true	false
<i>push(stk', i')</i>	false	$i=i'$ and $\text{equal}(\text{stk}, \text{stk}')$

Questa tabella può essere vista come l'aggiunta di una terza dimensione alla tabella di base per le osservazioni unarie.

Osservazione binaria: $\text{equal}(\text{stk}', m)$

Questa terza dimensione può essere rimossa andando a classificare esplicitamente solo il primo argomento dell'osservazione e referenziando in modo astratto il secondo argomento mediante una **variabile libera**.

<i>osservazione</i>	<i>Costruttore di stk'</i>	
	<i>newstack</i>	<i>push(stk, i)</i>
<i>equal(stk', m)</i>	isnew(m)	not isnew(m) and i=top(m) and equal(stk, pop(m))

Specifica algebrica di coda

Specifica sintattica

sorts: queue, item, boolean

operations:

newq() \rightarrow queue

addq(queue, item) \rightarrow queue

deleteq(queue) \rightarrow queue

frontq(queue) \rightarrow item

isnewq(queue) \rightarrow boolean

Richiamo: Specifica assiomatica di *Coda*

Specifica semantica

Operatori:

CREACODA() $=Q'$

Post: $Q' = \wedge$, la sequenza vuota

CODAVUOTA(Q) $=b$

Post: $b = \text{vero}$, se $Q = \wedge$; $b = \text{falso}$, altrimenti

LEGGICODA(Q) $=a$

Pre: $Q = a_1, a_2, \dots, a_n$ e $n \geq 1$

Post: $a = a_1$

FUORICODA(Q) $=Q'$

Pre: $Q = a_1, a_2, \dots, a_n$ e $n \geq 1$

Post: $Q' = a_2, \dots, a_n$ se $n > 1$; $Q' = \wedge$ se $n = 1$

INCODA(a,Q) $=Q'$

Pre: $Q = a_1, a_2, \dots, a_n$ e $n \geq 0$

Post: $Q' = a_1, a_2, \dots, a_n, a$ se $n > 0$; $Q' = a$ se $n = 0$

Costruttori e Osservazioni

<i>osservazioni</i>	<i>Costruttore di q'</i>	
	<i>newq</i>	<i>addq(q, i)</i>
<i>isnew(q')</i>	true	false
<i>frontq(q')</i>	error	if isnewq(q) then i else frontq(q)
<i>deleteq(q')</i>	error	if isnewq(q) then newq else addq(delete(q), i)

Specifica algebrica di coda

Specifica semantica

declare **q,r**: queue, **i**: item

isnewq(newq) = true

isnewq(addq(q, i)) = false

deleteq(addq(q, i)) = **if** isnewq(q) **then** newq **else** addq(deleteq(q),i)

frontq(addq(q, i)) = **if** isnewq(q) **then** i **else** frontq(q)

Specifica algebrica di coda

Specifica di restrizione

restrictions

frontq(newq) = error

deleteq(newq) = error

Esercizio

- Fornire una specifica algebrica completa, consistente e minimale (cioè non ridondante) del dato astratto *Vettore* di cui è stata fornita la specifica assiomatica mediante pre- e postcondizioni.

sorts: tipoelem, array, integer

operations:

new() -> array

read(array, integer) -> tipoelem

write(array, integer, tipoelem) -> array

N.B. Supponiamo che l'array sia inizializzato con il valore 0.

Esercizio (cont.)

<i>osservazioni</i>	<i>Costruttore di v'</i>	
	<i>new()</i>	<i>write(v, i, e)</i>
<i>read(v', i')</i>	0	if $i=i'$ then e else $\text{read}(v, i')$

Esercizio (cont.)

sorts: array, integer, tipoelem

create(integer, integer) -> array //crea un vettore con
lower/upper bound

assign(array, integer, tipoelem) -> array

first(array) -> integer // restituisce lower bound

last(array) -> integer // restituisce upper bound

eval(array, integer) -> tipoelem

operations:

first(create(x,y))=x

first(assign(a,n,e))first(a)

last(create(x,y))=y

last(assign(a,n,e))=last(a)

eval(create(x,y),n)=error

eval(assign(a,n,e),m)=if $m < \text{first}(a)$ or $m > \text{last}(a)$ then
error else if $m = n$ then e else eval(a,m)

Esercizio

- Fornire una specifica algebrica semantica completa, consistente e minimale per il dato astratto *Stringa* supposto che le specifiche sintattiche siano le seguenti:

sorts: string, char, integer, boolean

operations:

new() → string

crea nuove stringhe

append(string, string) → string

concatena due stringhe

add(string, char) → string

aggiunge un carattere a fine stringa

length(string) → integer

calcola la lunghezza di una stringa

isEmpty(string) → boolean

stabilisce se la stringa è vuota

equal(string, string) → boolean

stabilisce se due stringhe sono uguali

La scelta dei costruttori

Talvolta, l'applicazione della metodologia per la sintesi di specifiche algebriche può comportare delle difficoltà riguardo al **discernimento di cosa è costruttore e cosa operazione**.

Ad esempio, guardando la specifica sintattica del dato astratto *Stringa* possiamo dire che tutti gli operatori che restituiscono una stringa sono dei candidati a essere dei costruttori.

`new()` \rightarrow string

`append(string, string)` \rightarrow string

`add(string, char)` \rightarrow string

Mentre non ci sono dubbi sul primo, restano delle perplessità sul fatto che entrambi gli altri operatori debbano essere dei costruttori.

La scelta dei costruttori

Nello scegliere i costruttori adotteremo il **criterio di minimalità**, cioè l'insieme dei costruttori dev'essere il più piccolo insieme di operatori necessario a costruire tutti i possibili valori per un certo dato astratto.

Ora, è evidente che per costruire una stringa abbiamo bisogno di:

`new() → string` `add(string, char) → string`

Il terzo operatore:

`append(string, string) → string`

non è necessario, quindi **non** lo scegliamo come costruttore.

Un criterio euristico: *gli argomenti di un costruttore non devono essere tutti dello stesso sort del dato astratto.*

Esercizio (cont.)

<i>osservazioni</i>	<i>Costruttore di s'</i>	
	<i>new()</i>	<i>add(s, c)</i>
<i>length(s')</i>	0	length(s)+1
<i>isEmpty(s')</i>	true	false
<i>append(t,s')</i>	t	add(append(t,s),c)

Esercizio (cont.)

`equal(string, string) → boolean`

<i>Costruttore di t'</i>	<i>Costruttore di s'</i>	
	<i>new()</i>	<i>add(s, c)</i>
<i>new()</i>	true	false
<i>add(t, h)</i>	false	if h=c then equal(t, s) else false

Esercizio

Progettare il dato astratto *Conto Con Fido* che consente di rappresentare dei conti correnti bancari per i quali è permesso uno scoperto. Si deve poter limitare lo scoperto tramite concessione del fido. Le operazioni ammesse per questo dato astratto sono:

conto (account): apre un nuovo conto corrente bancario definendo saldo iniziale e massimo scoperto ammesso

saldo (balance): riporta il saldo del conto

deposita (deposit): deposita una somma sul conto

preleva (withdraw): preleva denaro dal conto

concediFido (setOverdraftLimit): definisce il limite massimo dello scoperto

fidoConcesso (getOverdraftLimit): restituisce il limite massimo dello scoperto

Esercizio (cont.)

Specifica sintattica

sorts: contocorrente, saldo, denaro, scoperto;

operations:

conto(saldo, scoperto) → contocorrente

saldo(contocorrente) → saldo

deposita(contocorrente, denaro) → contocorrente

preleva(contocorrente, denaro) → contocorrente

concediFido(contocorrente, scoperto) → contocorrente

fidoConcesso(contocorrente) → scoperto

Esercizio (cont.)

Specifica semantica

Il dominio *contocorrente* è l'insieme delle coppie $(s, l) \in \mathbf{R} \times \mathbf{R}^+$, tali che $s \geq -l$. Il dominio *saldo* corrisponde all'insieme dei numeri reali \mathbf{R} , mentre i domini *scoperto* e *denaro* corrispondono ad \mathbf{R}^+ .

Pertanto potremo disporre sui vari domini di tipiche operazioni algebriche, e in particolare:

- due operazioni binarie: “+” e “-”
- operatore unario: “-”

nonché di una relazione binaria “<” e di una costante “0”.

N.B.: formalmente avremmo dovuto indicare che “+” è definito sui domini *saldo* e *scoperto* e restituisce un *saldo*, etc.,

Esercizio (cont.)

N.B.: Gli operatori aggiuntivi che sono stati specificati non fanno riferimento al dominio *contocorrente* relativo al dato astratto che si vuole definire. Per questo non è necessario specificarli.

Diversamente, se fosse stata ipotizzata la disponibilità di un operatore del dominio *contocorrente*, avremmo dovuto successivamente indicarne la semantica al pari degli altri operatori indicati nella specifica sintattica.

Esercizio (cont.)

Possiamo identificare un solo costruttore (*conto*). Gli altri operatori sono tutte osservazioni.

<i>osservazioni</i>	<i>Costruttore di c</i>
	<i>conto(s,l)</i>
<i>saldo(c)</i>	s
<i>deposita(c,d')</i>	conto(s+d',l)
<i>preleva(c,d')</i>	if s-d'<-l then error else conto(s-d',l)
<i>concediFido(c,l')</i>	if s<-l' then error else conto(s,l')
<i>fidoConcesso(c)</i>	l

Esercizio (cont.)

Specifica semantica

declare **c**: contocorrente, **s**: saldo, **d**: denaro, **l**, **l'**: scoperto;

saldo(conto(s,l))=s

deposita(conto(s,l), d) = conto(s+d,l)

preleva(conto(s,l), d) = **if** s-d<-l **then** error **else** conto(s-d,l)

concediFido(conto(s,l),l') = **if** s<-l' **then** error **else** conto(s,l')

fidoConcesso(conto(s,l))=l

Esercizio (cont.)

Utilizzando le precedenti specifiche è possibile dimostrare la seguente equazione:

$$\text{preleva}(\text{deposita}(c, d), d) = c$$

Dim.: Sia $c = \text{conto}(s, l)$

$$\text{preleva}(\text{deposita}(\text{conto}(s, l), d), d) = \text{preleva}(\text{conto}(s+d, l), d)$$

per definizione di *deposita* ($\text{deposita}(\text{conto}(s, l), d) = \text{conto}(s+d, l)$)

Inoltre:

$$\text{preleva}(\text{conto}(s+d, l), d) = \text{conto}((s+d)-d, l)$$

per definizione di *preleva* ($\text{preleva}(\text{conto}(s, l), d) = \text{if } s-d < -l \text{ then error else } \text{conto}(s-d, l)$).

$$\text{Ma } \text{conto}((s+d)-d, l) = \text{conto}(s, l) = c$$

Esempio: dato astratto *MazzoCarte*

Progettare il dato astratto *Mazzo Carte* che consente di gestire un mazzo di carte da gioco. L'insieme di operazioni ammesse è il seguente:

creaMazzo: restituisce un mazzo di carte vuoto

aggiungiCarta: aggiunge una carta in cima al mazzo

carteNelMazzo: conta il numero di carte nel mazzo

daiCarta: restituisce la carta in cima al mazzo

fineMazzo: verifica che il mazzo sia vuoto o meno

mescolaMazzo: mescola le carte del mazzo

Dato astratto MazzoCarte

Specifica sintattica

sorts mazzo, carta, boolean, integer;

- `creaMazzo()` \rightarrow mazzo
- `aggiungiCarta(mazzo, carta)` \rightarrow mazzo
- `carteNelMazzo(mazzo)` \rightarrow integer
- `scopriCarta(mazzo)` \rightarrow carta
- `fineMazzo(mazzo)` \rightarrow boolean
- `mescolaMazzo(mazzo)` \rightarrow mazzo

Dato astratto MazzoCarte

Il dominio *mazzo* è l'insieme delle possibili sequenze, inclusa quella vuota, di carte. Ipotezziamo che sul dominio *integer*, oltre all'operazione di addizione, sia definita una funzione

random(integer) → integer

che preso in input un numero intero *n*, genera causalmente un intero compreso fra 1 e *n+1*.

Per definire agevolmente la semantica dell'operatore *mescolaMazzo* necessitiamo di un ulteriore operatore:

aggiungiCartaACaso(mazzo, carta) → mazzo

che aggiunge una carta in una posizione casuale nel mazzo.

Questo operatore è solo di ausilio alla definizione di *mescolaMazzo* e non è visibile dal fruitore del dato astratto.

MazzoCarte (spec. semantica)

Specifica semantica

declare m, m' : mazzo, c, c' : carta;

<i>osservazioni</i>	<i>Costruttore di m'</i>	
	<i>creaMazzo</i>	<i>aggiungiCarta(m, c)</i>
<i>carteNelMazzo(m')</i>	0	<i>carteNelMazzo(m)+1</i>
<i>scopriCarta(m')</i>	error	c
<i>fineMazzo(m')</i>	true	false
<i>mescolaMazzo(m')</i>	creaMazzo	<i>aggiungiCartaACaso(mescolaMazzo(m),c)</i>
<i>aggiungiCartaACaso(m', c')</i>	<i>aggiungiCarta(creaMazzo, c')</i>	<i>if random(carteNelMazzo(m')) = carteNelMazzo(m')+1 then aggiungiCarta(m', c') else aggiungiCarta(aggiungiCartaACaso(m, c'), c)</i>

MazzoCarte (spec. Semantica)

Utilizzando le precedenti specifiche è possibile dimostrare la seguente equazione:

$$\text{mescolaMazzo}(\text{aggiungiCarta}(\text{creaMazzo}, c)) = \text{aggiungiCarta}(\text{creaMazzo}, c)$$

cioè mescolando un mazzo di una carta si ottiene il mazzo di carte iniziale.

Dim.:

$$\begin{aligned} \text{mescolaMazzo}(\text{aggiungiCarta}(\text{creaMazzo}, c)) &= \\ &= \text{aggiungiCartaACaso}(\text{mescolaMazzo}(\text{creaMazzo}), c) = \\ &= \text{aggiungiCartaACaso}(\text{creaMazzo}, c) \end{aligned}$$

per definizione di *mescolaMazzo*. Inoltre:

$$\text{aggiungiCartaACaso}(\text{creaMazzo}, c) = \text{aggiungiCarta}(\text{creaMazzo}, c)$$

per definizione di *aggiungiCartaACaso*. c.v.d.

Esercizio

Estendere le specifiche del dato astratto *Mazzo Carte* con i seguenti operatori:

estraiCarta: restituisce un mazzo di carte privo della carta in cima.

estraiCartaACaso: restituisce un mazzo di carte privo di una carta qualunque (non necessariamente quella in cima).

èNelMazzo: controlla che una carta sia presente nel mazzo di carte

contenuto: controlla che un mazzo di carte sia insiemisticamente contenuto in un altro mazzo di carte (l'ordine delle carte non conta)

uguali: verifica che due mazzi di carte siano insiemisticamente identici (l'ordine delle carte non conta).

Esercizio (soluzione)

- Specifica sintattica
 - estraiCarta(mazzo) \rightarrow carta //carta in cima
 - estraiCartaACaso(mazzo) \rightarrow carta //carta a caso
 - èNelMazzo(mazzo, carta) \rightarrow boolean
 - contenuto(mazzo, mazzo) \rightarrow boolean
 - uguali(mazzo, mazzo) \rightarrow boolean

Esercizio (soluzione)

<i>osservazioni</i>	<i>Costruttore di m'</i>	
	<i>creaMazzo</i>	<i>aggiungiCarta(m,c)</i>
<i>estraiCarta(m')</i>	error	c
<i>estraiACaso(m')</i>	error	if random(carteNelMazzo(m))=carteNelMazzo(m') then c else estraiCartaACaso(m)
<i>èNelMazzo(m', c')</i>	false	if c'=c then true else èNelMazzo(m, c')

Esercizio (soluzione): *contenuto*

$\text{contenuto}(m', n')$ // m' contenuto in n' l'ordine non è importante

<i>costruttore n'</i>	<i>Costruttore di m'</i>	
	<i>creaMazzo</i>	<i>aggiungiCarta(m, c)</i>
<i>creaMazzo</i>	true	false
<i>aggiungiCarta(n, d)</i>	true	if $\text{èNelMazzo}(n', c)$ then $\text{contenuto}(m, n')$ else false

Esercizio (soluzione): *contenuto*

uguale(m' , n')

<i>costruttore n'</i>	<i>Costruttore di m'</i>	
	<i>creaMazzo</i>	<i>aggiungiCarta(m, c)</i>
<i>creaMazzo</i>	true	false
<i>aggiungiCarta(n, d)</i>	false	if contenuto(m' , n') then if contenuto(n' , m') then true else false else false

Astrazione di controllo

- L'astrazione di controllo si riferisce alla possibilità di specificare un modulo software che **esegue** delle operazioni in un **ordine**, nascondendo i dettagli su come il mantenimento dell'ordine è ottenuto
- In sintesi:
 - Il modulo software deve essere parametrizzato rispetto alle operazioni da eseguire
 - Il modulo software è associato a un controllo di sequenza (*control flow*)
 - I dettagli di come il controllo di sequenza è garantito non sono visibili al consumatore (fruitore) del modulo

Astrazione di controllo

- Un modulo che ha in input un'espressione e un comando e itera l'esecuzione del comando fintanto che l'espressione è vera, è un modulo che implementa il costrutto di controllo `while`
- L'effettiva implementazione del `while` è nascosta

Esempio: `while(espressione, istruzione)`

Può essere realizzato iterativamente in vari modi:

1. Valuta espressione
2. Se true allora vai a 4
3. Esci dal modulo (return)
4. Esegui istruzione
5. Vai a 1

1. Valuta espressione
2. Se false allora vai a 5
3. Esegui istruzione
4. Vai a 1
5. Esci dal modulo (return)

Astrazione di controllo

Esempio (cont.): o persino ricorsivamente!

1. Valuta espressione
2. Se true allora
 - a) Esegui istruzione
 - b) `while(espressione, istruzione).`
3. Esci dal modulo (`return`)

Tutte queste realizzazioni (iterative o ricorsive) sono corrette rispetto alla semantica dell'astrazione di controllo `while`. All'utente del modulo non importa sapere quale delle realizzazioni è stata adottata.

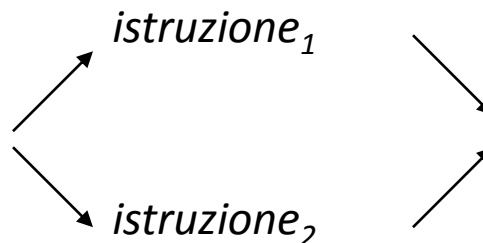
Astrazione di controllo

Nella programmazione sequenziale l'ordine di esecuzione delle istruzioni è **totale**. Nella programmazione concorrente/parallela l'ordine di esecuzione può essere **parziale**.

Esempio: l'astrazione di controllo

fork-join(istruzione₁, istruzione₂)

che prevede l'esecuzione parallela di *istruzione₁* e *istruzione₂* definisce una relazione d'ordine parziale fra le due istruzioni.



Astrazione di controllo

I linguaggi di specifica utilizzati per l'astrazione di controllo si basano sulla definizione di **relazioni di precedenza** fra istruzioni che permettono di stabilire sia ordinamenti totali che ordinamenti parziali.

La combinazione dell'astrazione dati con l'astrazione di controllo permette di sviluppare soluzioni software **invarianti** non solo al cambiamento della realizzazione dei dati ma anche **al cambiamento dei meccanismi di visita del dato astratto**.

Ciò risulta molto utile poiché l'iterazione su collezioni di dati è un compito comune e ripetitivo e l'evidenza empirica indica che spesso si commettono errori nella inizializzazione dell'iterazione e nella scrittura delle condizioni di stop.

Astrazione di controllo

Esempio: Un dato astratto che **contiene** un insieme di valori può mettere a disposizione due operatori, **next** che restituisce un valore e il predicato **hasNext** che è vero se ci sono ancora altri valori da selezionare. L'astrazione di controllo *for-each(dato astratto, istruzione)* utilizzerà questi due operatori per eseguire l'*istruzione* su ogni valore contenuto nel dato astratto. **L'informazione sull'ordine con cui i valori sono considerati (cioè sul protocollo di iterazione) è nascosta all'utente del *for-each*.**

Andando ancora oltre, si può ipotizzare che il dato astratto deleghi un modulo **iteratore** a fornire i servizi richiesti dal *for-each*. In questo modo i meccanismi di visita sono tutti incapsulati in un modulo.

Ancora una volta, la tecnica dell'incapsulamento viene in aiuto al progettista che vuole applicare una qualche forma di astrazione.

Riferimenti bibliografici

Per la progettazione con astrazione funzionale e astrazione dati

Peter Klein

Designing Software with Modula-3

Per le specifiche assiomatiche di vettore, pila e coda

Alan Bertossi

Algoritmi e Strutture di Dati

UTET, 2000

Capitoli 0, 1, 4

Per le specifiche algebriche

John D. Gannon, James M. Purtilo, Marvin V. Zelkowitz

Software Specification: A Comparison of Formal Methods

2001

Capitolo 5.3

Riferimenti bibliografici

Per le specifiche algebriche si può fare riferimento anche al testo:

A. Fuggetta, C. Ghezzi, S. Morasca, A. Morzenti, M. Pezzè

Ingegneria del software: progettazione, sviluppo e verifica

Mondadori Informatica, 1991

Capitolo 4.3

Astrazione nella programmazione

Astrazione e linguaggi ...

- L'applicazione dei principi di astrazione nella fase progettuale sarebbe poco efficace se poi nella fase di realizzazione non fosse possibile continuare a nascondere l'informazione (trasformazioni, rappresentazioni, controlli di sequenza)
- Pertanto è indispensabile che i **linguaggi di programmazione**, che costituiscono il principale strumento utilizzato in fase di realizzazione, **supportino l'applicazione dei principi di astrazione** al fine di agevolare lo sviluppo e la manutenzione di sistemi software sempre più complessi

Astrazione e linguaggi ...

In linea di principio, ogni **linguaggio simbolico** offre di per se una forma di astrazione dalla particolare macchina.

Il fatto che una istruzione (e.g., LOAD) abbia un codice in linguaggio macchina che sia

10011000

piuttosto che

11101101

è poco rilevante per il programmatore. Come sono spesso irrilevanti le locazioni di memoria che conterranno i risultati (parziali) di una computazione.

Astrazione e linguaggi ...

Per questa ragione a metà degli anni '50 vennero introdotti i nomi mnemonici per le istruzioni e per l'accesso alla memoria, dando vita ai primi linguaggi assembler.

Il principio dell'astrazione applicato nel progetto dei linguaggi assemblativi fu quello di nascondere i dettagli di codifica di istruzioni e indirizzi di memoria.

Ben presto ci si accorse di poter fare di meglio. Il programmatore assembler era costretto a conoscere la particolare rappresentazione di un dato (e.g., di un floating point, di un intero, di un vettore, etc.). Inoltre c'erano delle strutture di controllo ricorrenti, per esempio l'iterazione, che occorreva ogni volta implementare in qualche modo.

Astrazione e linguaggi ...

Alla fine degli anni '50 vennero introdotti i primi **linguaggi ad alto livello**, caratterizzati dalla possibilità di trattare particolari domini di valori (e.g., interi, reali, caratteri, numeri complessi) indipendentemente dalla loro rappresentazione in memoria centrale (I+J in Fortran, indicava la somma di due interi, indipendentemente da quanti bit fossero stati riservati ai due interi).

I linguaggi di programmazione di alto livello introdussero anche delle strutture di controllo astratte (come for, while, etc.) e dei nuovi operatori (e.g., moltiplicazione) non previsti a livello di linguaggio macchina.

Principio di astrazione...

Riepilogando, negli anni '50 e '60 si introdussero diverse forme di astrazione nei linguaggi di programmazione (strutture di controllo, operatori, dati).

Si giunse quindi a maturare la convinzione che:

“È possibile costruire astrazioni su una qualunque classe sintattica, purché le frasi di quella classe specifichino un qualche tipo di computazione”

...Principio di astrazione

Di seguito si esaminerà l'applicazione del principio di astrazione a sei classi sintattiche, in particolare:

- Espressione → astrazione di funzione
- Comando → astrazione di procedura
- Controllo di sequenza → astrazione di controllo
- Accesso a un'area di memoria → astrazione di selettore
- Definizione di un dato → astrazione di tipo
- Dichiarazione → astrazione generica

Tutte queste classi sintattiche sottintendono una computazione ...

...Principio di astrazione.

Infatti, ...

- L'astrazione di una *funzione* include un'espressione da valutare
- L'astrazione di una *procedura* include un comando da eseguire
- L'astrazione di *controllo* include delle espressioni di controllo dell'ordine di esecuzione delle istruzioni
- L'astrazione di *selettore* include il calcolo dell'accesso ad una variabile
- L'astrazione di *tipo* include un gruppo di operatori che definiscono implicitamente un insieme di valori
- L'astrazione *generica* include una frase che sarà elaborata per produrre legami (binding)

Controesempio:

La classe sintattica “letterale” non può essere astratta perché non specifica alcun tipo di computazione.

Astrazione di funzione

Un'astrazione di funzioni include un'espressione da valutare, e quando chiamata, darà un valore come risultato.

Un'astrazione di funzione è specificata mediante una definizione di funzione, del tipo:

function I(FP₁; ...; FP_n) is E

dove **I** è un identificatore, **FP₁; ...; FP_n** sono parametri formali, ed **E** è l'espressione da valutare.

In questo modo si lega **I** a un'entità, l'astrazione di funzione, che ha la proprietà di dare un risultato ogni qualvolta è chiamata con argomenti appropriati.

Astrazione di funzione

Una chiamata di funzione, $I(AP_1; \dots; AP_n)$ dove i parametri effettivi, $AP_1; \dots; AP_n$ determinano gli argomenti, ha due punti di vista:

- Punto di vista dell'*utente*: la chiamata di una funzione trasforma gli argomenti in un risultato;
- Punto di vista dell'*implementatore*: la chiamata valuta E , avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti. L'algoritmo codificato in E è di interesse per l'implementatore.

Anomalie di progetto dei linguaggi

Se l'astrazione di funzione include un'espressione **E** da valutare, è naturale attendersi che il corpo della funzione non includa dei comandi (come le assegnazioni, le istruzioni di salto, le iterazioni, ecc.) il cui effetto è quello di cambiare lo stato di un sistema, non di produrre valori.

Non sempre ciò accade, come si può osservare in questo esempio di funzione scritta in Pascal.

```
function power(x:real;n:integer) : real
begin
  if n= 1 then power := x
  else power := x*power(x,n-1)
end
```

Anomalie di progetto dei linguaggi

- Nel corpo della funzione Pascal compaiono i **comandi di assegnazione**.
- Non è proprio possibile farne a meno, perché per poter restituire un valore, in Pascal è necessario assegnare un valore a una pseudo variabile, che ha lo stesso nome della funzione.
- Nell'esempio: l'identificatore della funzione, **power**, può denotare sia l'espressione da valutare che la pseudo variabile dove sarà depositato il risultato.
- Il corpo della funzione è sintatticamente un comando, ma semanticamente è una espressione (la funzione, infatti può essere invocata solo alla destra di operazioni di assegnazione).

Anomalie di progetto dei linguaggi

- Comandi nel corpo di funzioni. Necessariamente così?
- No. Riportiamo di seguito un esempio nel linguaggio funzionale ML.

```
function power(x:real;n:int) is  
  if n= 1  then x  
  else x*power(x,n-1)
```

Nel corpo della funzione definita in ML compare una espressione condizionale la cui valutazione non modifica lo stato del sistema.

Anomalie di progetto dei linguaggi

- Ma allora perché Pascal, Ada e altri linguaggi di programmazione permettono di avere comandi nel corpo di funzioni?
- Per poter sfruttare la potenza espressiva dell'assegnazione e dell'**iterazione** nella computazione di risultati.
- Diversamente saremmo costretti a esprimere ricorsivamente le espressioni da valutare. E la ricorsione, com'è noto, può essere fonte di inefficienze nell'uso delle risorse di calcolo.

Anomalie di progetto dei linguaggi

Morale

Molti linguaggi permettono di avere comandi nel corpo di funzioni. Questo per ragioni di efficienza.

Si lascia al programmatore il compito di utilizzare correttamente i comandi in modo da evitare che la funzione possa avere effetti collaterali (*side effect*) oltre quello di calcolare un valore.

Astrazione di funzione

Le funzioni, in quanto astrazioni di espressioni, possono comparire ovunque occorra un'espressione.

Pertanto, esse compaiono alla destra di operazioni di assegnazioni, ma anche al posto di parametri effettivi nelle chiamate di altre funzioni (o procedure), laddove un valore potrebbe essere calcolato mediante una espressione.

$$y := f(y, \text{power}(x, 2))$$

La funzione f ha due parametri passati per valore.

Astrazione di funzione

In molti linguaggi di programmazione è possibile definire dei parametri formali che sono un riferimento a una funzione.

Esempio: in Pascal

```
function Sommatoria(function F(R: real, M: integer): real; X: real;  
    N:integer): real;  
var I: integer;  
    Sum: real;  
begin  
    Sum := 0;  
    for I := 1 to N do Sum := Sum + F(x, I);  
    Sommatoria := Sum  
end
```

Astrazione di funzione

Esempio (cont.)

Y := Sommatoria(**power**, x, n);

calcola la sommatoria: $\sum_{i=1}^n x^i$

In alcuni linguaggi di programmazione si separa il concetto di astrazione di funzione da quello di legame (binding) a un identificatore. Questo permette di passare come parametri delle funzioni anonime (prive di identificatore).

Infine, le funzioni possono essere il **risultato** di valutazioni di espressioni o possono essere assegnate a variabili:

val cube = fn (x: real) => x*x*x

← in ML

Astrazione di funzione

Riepilogando:

In alcuni linguaggi di programmazione (Fortran, Ada-83) le funzioni sono di **terza classe**, cioè possono essere solo chiamate, in altri (vedi Pascal) sono di **seconda classe**, cioè possono essere passate come argomenti, mentre in altri linguaggi di programmazione esse sono di **prima classe** in quanto possono essere anche restituite come risultato della chiamata di altre funzioni o possono essere assegnate come valore a una variabile (alcuni linguaggi di programmazione funzionali, come Lisp e ML, e linguaggi di scripting, come Perl, permettono di generare funzioni al run-time).

Cittadini di prima classe

In generale, in programmazione una qualunque entità (dato, procedura, funzione, etc.) si dice **cittadino di prima classe** quando non è soggetta a restrizioni nel suo utilizzo.

Cittadini di prima classe

A tal proposito, osserviamo che i valori possono essere:

- *Denotabili*, se possono essere associati ad un nome;
- *Esprimibili*, se possono essere il risultato di un'espressione complessa (cioè diversa da un semplice nome)
- *Memorizzabili*, se possono essere memorizzati in una variabile.

Cittadini di prima classe

Esempio:

Nei linguaggi imperativi, i valori di tipo intero sono in genere sia denotabili, che esprimibili che memorizzabili.

Al contrario, i valori del tipo delle funzioni da Integer a Integer sono denotabili in quasi tutti i linguaggi, perché possiamo dare loro un nome con una dichiarazione:

```
int succ(int x) { return x+1 }
```

ma **non sono esprimibili** in quasi tutti i linguaggi imperativi, perché non ci sono espressioni complesse che restituiscono una astrazione di funzione come risultato della loro valutazione. Allo stesso modo **non sono valori memorizzabili**, perché non possiamo assegnare una funzione ad una variabile.

Cittadini di prima classe

Esempio (cont.)

La situazione è diversa per i linguaggi di altri paradigmi, quali ad esempi i linguaggi funzionali (Scheme, ML, haskell, ecc.), nei quali i valori funzionali sono sia denotabili, che esprimibili, che, in certi linguaggi, memorizzabili.

Stessa cosa dicasi per gli **oggetti** ai quali si accennerà in seguito. Nei linguaggi imperativi essi sono denotabili ma non esprimibili e memorizzabili. In altri termini, gli oggetti non sono cittadini di prima classe. Al contrario, nei linguaggi orientati agli oggetti, gli oggetti sono denotabili, esprimibili e memorizzabili, cioè sono cittadini di prima classe.

Astrazione di procedura

Un'astrazione di procedura include un **comando** da eseguire, e quando chiamata, aggiornerà le variabili che rappresentano lo stato del sistema.

Un'astrazione di procedura è specificata mediante una definizione di procedura, del tipo:

procedure $I(FP_1; \dots; FP_n)$ is C

dove **I** è un identificatore, **$FP_1; \dots; FP_n$** sono parametri formali, e **C** è il blocco di comandi da eseguire.

In questo modo si lega **I** all'astrazione di procedura, che gode della proprietà di cambiare lo stato del sistema quando chiamata con argomenti appropriati.

Astrazione di procedura

Data una chiamata di procedura, $I(AP_1; \dots; AP_n)$ dove $AP_1; \dots; AP_n$ sono i parametri effettivi, il punto di vista dell'*utente* è che la chiamata aggiornerà lo stato del sistema in modo dipendente dai parametri, mentre il punto di vista dell'*implementatore* è che la chiamata consentirà l'esecuzione del corpo di procedura **C**, avendo precedentemente vincolato i parametri formali agli argomenti corrispondenti. L'algoritmo codificato in **C** è di interesse solo per l'implementatore.

Astrazione di procedura

Esempio di puntatori a funzioni in C

La funzione **bubble** ordina un array di interi sulla base di una funzione di ordinamento.

L'argomento con puntatore a funzione

```
void (*compare) ( int, int, int *)
```

Dice a **bubble** di aspettarsi un puntatore a una funzione, identificata da **compare**, che prende tre argomenti in ingresso e restituisce un tipo **void**.

N.B.: Se avessimo rimosso le parentesi

```
void *compare( int, int, int * )
```

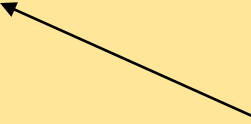
avremmo dichiarato semplicemente una funzione che prende in input tre interi e restituisce un puntatore a **void**.

```

1/* Fig. 7.26: fig07_26.c
2 Multipurpose sorting program using function pointers */
3 #include <stdio.h>
4 #define SIZE 10
5 void bubble( int [], const int, void (*)( int, int, int * ) );
6 void ascending( int, int, int * );
7 void descending( int, int, int * );
8
9int main()
10 {
11
12     int order,
13         counter,
14         a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16     printf( "Enter 1 to sort in ascending order,\n"
17           "Enter 2 to sort in descending order: " );
18     scanf( "%d", &order );
19     printf( "\nData items in original order\n" );
20
21     for ( counter = 0; counter < SIZE; counter++ )
22         printf( "%5d", a[ counter ] );
23
24     if ( order == 1 ) {
25         bubble( a, SIZE, ascending );
26         printf( "\nData items in ascending order\n" );
27     }
28     else {
29         bubble( a, SIZE, descending );
30         printf( "\nData items in descending order\n" );
31     }
32

```

Si noti il parametro di tipo puntatore a funzione.



```

33  for ( counter = 0; counter < SIZE; counter++ )
34      printf( "%5d", a[ counter ] );
35
36  printf( "\n" );
37
38  return 0;
39 }
40
41 void bubble( int work[], const int size,
42             void (*compare)( int, int, int * ) )
43 {
44     int pass, count, bool;
45
46     void swap( int *, int * );
47
48     for ( pass = 1; pass < size; pass++ )
49
50         for ( count = 0; count < size - 1; count++ )
51             (*compare)( work[ count ], work[ count + 1 ], &bool);
52             if ( bool )
53                 swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58     int temp;
59
60     temp = *element1Ptr;
61     *element1Ptr = *element2Ptr;
62     *element2Ptr = temp;
63 }
64

```

ascending e descending restituiscono in **bool** il risultato che può essere 0 o 1. **bubble** chiama **swap** se **bool** è 1.

Nota come i puntatori a funzione sono chiamati usando l'operatore di dereferenziazione (*), che seppur non richiesto, enfatizza come **compare** sia un puntatore a funzione e non una funzione.

```

65 void ascending( int a, int b, int * c )
66 {
67     *c = b < a;    /* swap if b is less than a */
68     return
69 }
70 void descending( int a, int b, int * c )
71 {
72     *c = b > a;    /* swap if b is greater than a */
73     return }

```

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order

2 6 4 8 10 12 89 68 45 37

Data items in ascending order

2 4 6 8 10 12 37 45 68 89

Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order

2 6 4 8 10 12 89 68 45 37

Data items in descending order

89 68 45 37 12 10 8 6 4 2

Astrazione funzionale

L'astrazione di funzione o di procedura sono *due tecniche di programmazione* di supporto all'**astrazione funzionale**, intesa come *tecnica di progettazione del software* secondo la quale occorre distinguere la specifica di un operatore (come esso è visto e manipolato dall'utente) dalla sua realizzazione.

Quale sia la tecnica di programmazione (astrazione di funzione o astrazione di procedura) più opportuna da adottare dipende da:

- **Tipo di operatore progettato:**
 - ha effetti collaterali → astrazione di procedura
 - non ha effetti collaterali → astrazione di funzione

Astrazione funzionale

- Limiti imposti dal linguaggio di programmazione.

Esempio: se l'operatore *bubblesort* non modifica l'array passato in ingresso, ma ne restituisce uno ordinato, sarebbe più opportuno implementarlo ricorrendo a un'astrazione di funzione, in quanto restituisce un nuovo valore che è un intero array ordinato. Tuttavia, non tutti i linguaggi di programmazione permettono di definire funzioni che restituiscono dati qualsivoglia complessi. In alcuni linguaggi (vedi Pascal) gli array sono cittadini di seconda classe. In tal caso l'operatore *bubblesort* dovrà essere necessariamente realizzato mediante astrazione di procedura.

Astrazione di controllo

L'astrazione di controllo si applica alla classe sintattica struttura di controllo.

Le strutture di controllo definiscono l'ordine in cui le singole istruzioni o i gruppi di istruzioni (unità di programma) devono essere eseguiti.

Il linguaggio macchina generalmente fornisce due semplici meccanismi per governare il flusso di controllo delle istruzioni singole: **l'elaborazione in sequenza** e il **salto**.

Nei linguaggi assemblativi le istruzioni da eseguire in sequenza sono scritte l'una dopo l'altra (sottintendendo che vanno scritte in locazioni di memoria contigue); il salto è rappresentato da un'istruzione di jump:

jump to <indirizzo simbolico o label>

Astrazione di controllo

L'indirizzo simbolico è comunque un dettaglio di scarsa importanza per il programmatore: quello che conta è poter indicare le prossime istruzioni da eseguire.

Per questa ragione i linguaggi di alto livello hanno introdotto strutture di controllo astratte come la **selezione**:

if cond then S1	jump on <cond> to A
else S2	S2
	jump to B
	A: S1
	B: ...

Astrazione di controllo

Similmente sono state introdotte diverse strutture di controllo astratte per l'**iterazione**.

Inoltre l'utilizzo dello stack per conservare gli indirizzi di ritorno dalle chiamate di funzione/procedura si è tradotta nella possibilità di effettuare chiamate **ricorsive**.

Astrazione di controllo

Di particolare interesse è l'attuale tendenza a offrire strutture di controllo iterative per quei dati astratti che sono collezioni omogenee (e.g., insiemi, multiinsiemi, liste, array) di valori.

La struttura di controllo iterativa ha due parametri, il dato astratto e la variabile alla quale assegnare un valore contenuto nel dato astratto. Il meccanismo di visita della collezione di valori è incapsulato nel dato astratto, che dispone di operazioni lecite per consentire l'iterazione sulla collezione di oggetti.

Astrazione di controllo

Esempio: In Java è disponibile l'astrazione di controllo **for-each**, che permette di iterare su una collezione.

```
List<Integer> list = new LinkedList<Integer>();  
... /* si inseriscono degli elementi  
for (Integer n : list) {  
    System.out.println(n);  
}
```

L'operazione messa a disposizione di **LinkedList** per l'astrazione di controllo **for-each** è **iterator()**.

Astrazione di controllo

Se guardiamo alle strutture di controllo come espressioni che, quando valutate, definiscono l'ordine in cui eseguire dei comandi, possiamo specificare un'astrazione di controllo come segue:

control I(FP₁; ...; FP_n) is S

dove **I** è un identificatore di una nuova struttura di controllo, **FP₁; ...; FP_n** sono parametri formali, e **S** è una espressione di controllo che definisce un ordine di esecuzione.

Astrazione di controllo

Esempio: (in un ipotetico linguaggio di programmazione)

`control swap(boolean: cond, statement: S1,S2) is`

```
if cond then
  begin S1;S2 end
else
  begin S2;S1 end
endif
```

Argomento di tipo
statement



Argomento di tipo
espressione

La chiamata `swap(i<j, {i:=j}, {j:=i})` porta i al valore di j se i è minore di j, altrimenti porta j al valore di i.

Astrazione di controllo

Esempio: (in un ipotetico linguaggio di programmazione)

`control alt_execution(statement: S1,S2,S3) is`

`if abnormal (S1) then`

`S2`

`else`

`S3`

`endif`



Argomento di tipo
statement

abnormal è un predicato che ha in input un comando S1 e restituisce *true* se l'esecuzione di S1 è terminata in modo anomalo (è stata sollevata un'eccezione), *false* altrimenti.

Astrazione di controllo

I linguaggi di programmazione moderni mettono a disposizione dei meccanismi sofisticati per gestire le situazioni eccezionali come:

- Errori aritmetici (e.g., divisione per 0) o di I/O (e.g., leggi un intero ma ottieni un carattere)
- Fallimento di precondizioni (e.g., prelievo da una coda vuota)
- Condizioni imprevedibili (e.g., lettura di un fine file)

Quando viene sollevata un'eccezione, essa dev'essere catturata e gestita.

Astrazione di controllo

Tipici approcci per **localizzare il gestore dell'eccezione** sono:

- Cerca un gestore nel blocco (o sottoprogramma) corrente
- Se non c'è, forza l'uscita dall'unità corrente e solleva l'eccezione nell'unità chiamante
- Continua a risalire la gerarchia delle unità chiamanti finché non trovi un gestore oppure non raggiungi il livello più alto (main).

Cosa accade quando il gestore è trovato e l'eccezione è trattata?

- Si riparte dall'unità contenente il gestore (**modello di terminazione**), come in Ada.
- Si ritorna ad eseguire il comando che ha generato l'eccezione (**modello di ripresa**, *resumption*), come in PL/I

Astrazione di controllo

- I linguaggi di programmazione sequenziale utilizzano la sequenza, la selezione e la ripetizione per definire un **ordinamento totale** sulla esecuzione dei comandi in un programma.
- I linguaggi di programmazione paralleli utilizzano ulteriori costrutti del flusso di controllo, come **fork**, **cobegin**, o cicli **for** paralleli, per introdurre in modo esplicito un **ordinamento parziale** sulla esecuzione dei comandi.
- Dato che il parallelismo è una forma di controllo della sequenza di esecuzione, l'astrazione di controllo è particolarmente importante nella programmazione parallela.

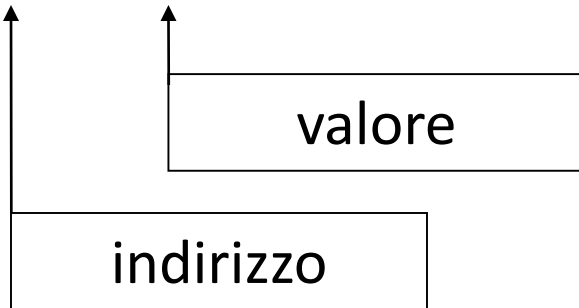
Astrazione di selettore

- Un linguaggio di programmazione dispone di costrutti per poter accedere ad una variabile (strutturata e non). Ad esempio in Pascal abbiamo:

```
var r: real;
```

...

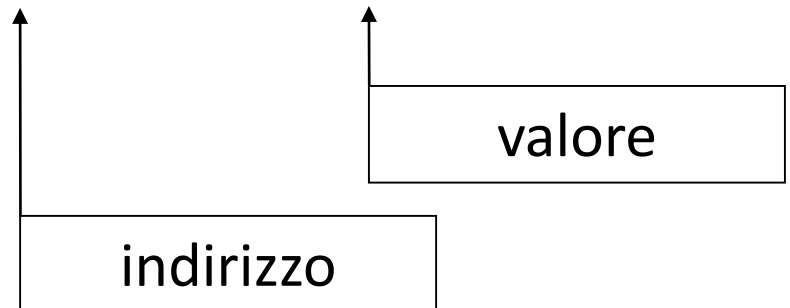
```
r := r*r;
```



```
var vett: array[1..10] of  
char;
```

...

```
vett[i] := vett[i+1];
```



Astrazione di selettore

- Lo stesso identificatore può essere usato sia come nome di un valore (legame dinamico fra identificatore e valore) e sia come un indirizzo (legame statico fra identificatore e locazione di memoria). Identicamente possiamo avere:

```
type rec = record
    a: real;
    b: real
end

var r: rec;

...

r.b := r.a*3;
```

Un accesso a una variabile restituisce un *riferimento* a una variabile.

Astrazione di selettore

- Tuttavia, se osserviamo il seguente codice Pascal:

```
type queue= ...;  
var Aq : queue;  
function first(q:queue):  
integer  
... (* restituisce il primo  
intero della coda *)  
...  
i := first(Aq);
```

ci accorgiamo che la chiamata `first(Aq)` può comparire solo alla destra di una assegnazione, perché le funzioni restituiscono valori, mentre a sinistra:

`first(Aq):=0;`

dovrebbe restituire riferimenti ad aree di memoria.

Astrazione di selettore

In Pascal abbiamo dei selettori predefiniti dal progettista del linguaggio:

- F^{\wedge} : riferimento a un puntatore F
- $V[E]$: riferimento a un elemento di un array V
- $R.A$: riferimento a un elemento di un record

ma il programmatore non ha modo di definire un nuovo selettore, come il riferimento a una lista di elementi indipendentemente da come la lista è realizzata.

In altri termini, non c'è la possibilità di definire astrazioni di selettore, che restituiscano l'accesso a un'area di memoria.

Astrazione di selettore

Per poter scrivere una assegnazione del tipo:

```
first(Aq) := 0;
```

dobbiamo poter definire un tipo di astrazione che quando chiamata restituisce il riferimento a una variabile (astrazione di selettore).

Supponiamo di estendere il Pascal con le astrazioni di selettore:

selector $l(FP_1; \dots; FP_n)$ **is** A

dove A è una espressione che restituisce un accesso a una variabile (che denoteremo con $\&$ come in C).

Astrazione di selettore

Potremo allora definire **first** come segue:

```
type queue= record
```

```
  elementi: array[1..max] of integer;
```

```
    testa, fondo, lung: 0..max;
```

```
  end;
```

```
selector first(q:queue) is &(q.elementi[q.testa]);
```

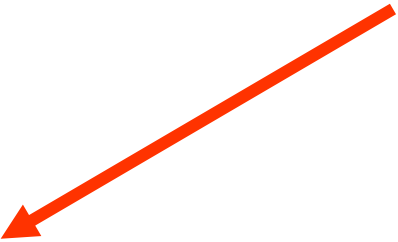
Questo ci consentirebbe di scrivere espressioni come:

```
first (Aq) :=first (Aq) +1 ;
```

dove l'invocazione di destra si riferisce alla funzione mentre quella di sinistra si riferisce all'astrazione di selettore.

Astrazione di selettore in C++

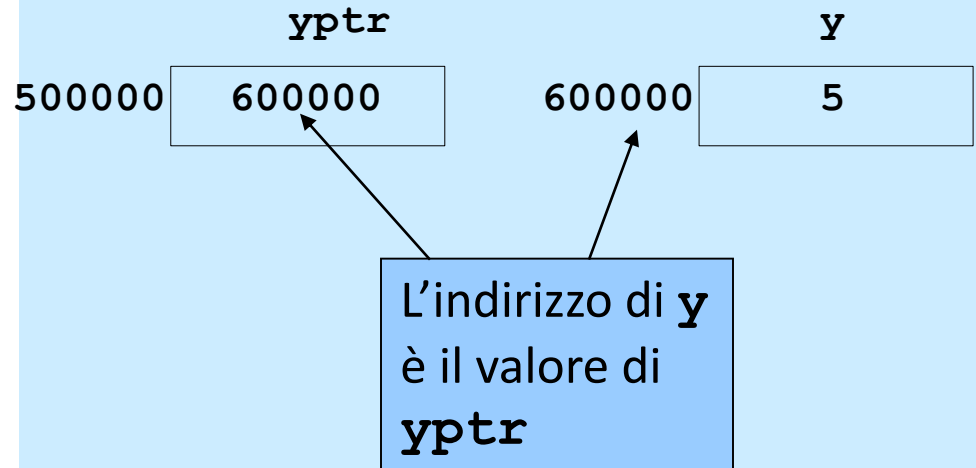
```
class Queue {  
...  
public:  
    static int & first(Queue *);  
};  
  
Queue q*;  
...  
  
int i=first(q);  
first(q) *= first(q);  
first(q)++;
```



& (operatore di indirizzamento)

- Restituisce l'indirizzo di un operando

```
int y = 5;  
int *yPtr;  
yPtr = &y;
```



Astrazione di selettore in C++

Una possibile implementazione:

```
class Queue {  
    int items[100];  
    int front, rear;  
public:  
    ...  
    int & first(Queue *q) {return q->items[front];}  
};
```

I dettagli dell'implementazione non sono di interesse per l'utente. Queue avrebbe potuto essere implementata diversamente.

Flessibilità dei linguaggi

L'espressività di un linguaggio di programmazione dipende direttamente da:

- I *meccanismi di composizione*, ossia la definizione di operazioni complesse a partire da quelle semplici
- I *meccanismi di controllo di sequenza*, ossia la possibilità di stabilire ordini di esecuzione in modo semplice
- I *valori* che il linguaggio permette di rappresentare e manipolare come dati

I linguaggi che supportano ...

- l'astrazione funzionale sono flessibili nei meccanismi di composizione
- l'astrazione di controllo sono flessibili nei meccanismi di controllo di sequenza
- l'*astrazione dati* sono flessibili nella definizione e manipolazione di nuovi valori

Tecniche di programmazione a supporto dell'astrazione dati

Le tecniche di **programmazione** a supporto dell'astrazione dati, che è una tecnica di progettazione del software, sono due:

- Definizione di **tipi astratti**, cioè l'astrazione sulla classe sintattica tipo
- Definizione di **classi di oggetti**, cioè l'astrazione sulla dichiarazione di moduli dotati di stato locale

Tecniche di programmazione a supporto dell'astrazione dati

In entrambi i casi occorre poter **incapsulare** la rappresentazione del dato con le operazioni lecite. Tuttavia,

Tipo astratto → il modulo rende visibile all'utilizzatore sia un **identificatore di tipo** che degli operatori

Classe di oggetti → il modulo rende visibile all'utilizzatore solo gli **operatori**

Inoltre:

Tipo astratto T → i **valori** sono associati a variabili dichiarate di **tipo T**

Classe di oggetti C → i **valori** sono associati a oggetti ottenuti per **istanziamento della classe oggetti C**

Tipo concreto e tipo astratto

I linguaggi ad alto livello mettono a disposizione del programmatore un nutrito gruppo di tipi predefiniti, detti **concreti**. Essi si distinguono in *primitivi* o *semplici* (cioè, i valori associati al tipo sono atomici) e *composti* o *strutturati* (i cui valori sono ottenuti per composizione di valori più semplici)

Tuttavia un linguaggio di programmazione sarà tanto più espressivo quanto più semplice sarà per il programmatore definire dei *suo*i tipi di dato a partire dai tipi di dato concreti disponibili. I tipi definiti dall'utente (detti anche *user defined types*, UDT) sono anche detti **astratti**

Tipo concreto e tipo astratto

Tipi concreti

(messi a disposizione
del linguaggio)



Primitivi

(valori atomici)

Composti

(valori ottenuti per
composizione di
valori più semplici)

Tipi astratti

(definiti dall'utente)

Astrazione di tipo

L'**espressione di tipo** (spesso abbreviato con **tipo**) è il costrutto con cui alcuni linguaggi di programmazione consentono di definire un nuovo tipo.

Esempio: in Pascal

```
type Person = record  
  name: packed array[1..20] of char;  
  age: integer;  
  height: real  
end
```

In questo caso si stabilisce **esplicitamente** una **rappresentazione** per i valori del tipo *Person* e **implicitamente** gli **operatori** applicabili a valori di quel tipo.

Per forza di cose, gli operatori del tipo *Person* dovranno essere generici (come l'assegnazione). Non è possibile specificarli!⁷⁶

Astrazione di tipo

Una **astrazione di tipo** (o **tipo astratto di dato** o, ancora più semplicemente, **tipo astratto**) ha un corpo costituito da una espressione di tipo. Quando è valutata, l'astrazione di tipo stabilisce sia una rappresentazione per un insieme di valori e sia le operazioni ad essi applicabili.

Analogamente a quanto detto per le altre astrazioni, l'astrazione di tipo potrà essere specificata come segue:

type I(FP₁; ...; FP_n) is T

dove **I** è un identificatore del nuovo tipo, **FP₁; ...; FP_n** sono parametri formali, e **T** è una espressione di tipo che specificherà la rappresentazione dei dati di tipo **I** e le operazioni ad esso applicabili.

Astrazione di tipo

Analizziamo le astrazioni di tipo che alcuni linguaggi di programmazione (come C e Pascal) consentono di definire.

```
type complex = record  
    Re: real;  
    Im: real  
end
```

consente di:

1. stabilire che `complex` è un identificatore di tipo;
2. associare una rappresentazione a `complex` espressa mediante tipi concreti già disponibili nel linguaggio.

Le operazioni associate al nuovo tipo `complex` sono tutte quelle che il linguaggio ha già previsto per il tipo `record` (e.g., assegnazione e selezione di campi).

Astrazione di tipo

I **limiti** di questa astrazione di tipo *à la Pascal* sono:

1. Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.
2. **Violazione del requisito di protezione**: l'utilizzatore è consapevole della rappresentazione del tipo `complex` (sa che è un record e quali sono i campi) ed è in grado di operare mediante operatori non specifici del dato.
3. **L'astrazione di tipo non è parametrizzata** (la comunicazione con il contesto esterno non è ammessa).

Di seguito si mostreranno le necessarie estensioni del linguaggio per superare questi limiti.

Astrazione di tipo

Problema: Il programmatore **non** può definire nuovi operatori specifici da **associare** al tipo.

Il problema può essere risolto mediante un costrutto di programmazione che permette di **incapsulare**

1. rappresentazioni del dato
2. operatori leciti.

Questo costrutto di programmazione è il **modulo**, ovvero

un gruppo di componenti dichiarate, come tipi, costanti, variabili, funzioni e persino (sotto) moduli.

Introduciamo quindi un costrutto **module** per supportare l'incapsulamento.

```

module complessi
  type complex= record
    R:real;
    I: real;
  end;

  function RealPart(c:complex):real
  begin return(c.R) end;
  function ImmPart(c:complex):real
  begin return(c.I) end;
  function ConsCx(r,i:real):complex
  var c1: complex;
  begin
    c1.R := r;
    c1.I := I;
    return c1
  end;

```

Dichiarazione di
operazioni

Dichiarazione di tipo

```

function + (c1,c2:complex):complex
  var c3: complex;
  begin
    c3.R := c1.R+c2.R;
    c3.I := c1.I+c2.I;
    return c3
  end;

```

```

function * (c1,c2:complex):complex
  var c3: complex;
  begin
    c3.R := c1.R*c2.R -
    c1.I*c2.I;
    c3.I := c1.R*c2.I + c1.I*c2.R;
    return c3
  end;

end module.

```

I Moduli: Uso

Il modulo introdotto è una **unità di programma compilabile separatamente**. Infatti, tutte le informazioni necessarie al compilatore per tradurre il modulo in codice oggetto sono nel modulo stesso.

Per poter utilizzare le componenti dichiarate in un modulo all'interno di un'altra unità di programma (modulo o programma principale) può essere necessario esplicitare l'importazione mediante una clausola, tipo:

uses <nome modulo>

I Moduli:Uso

Così per poter utilizzare il tipo astratto *complex* in un programma si scriverà:

`uses complessi`

In questo modo si potranno dichiarare delle variabili:

`x,y,z: complex`

e utilizzare gli operatori definiti nel modulo per i numeri complessi:

`x:=ConsCx(1.0,2.0)`

`y:=ConsCx(1.0,2.5)`

`z:=x+y`

I Moduli:Uso

Di fatto la dichiarazione precedente considera sempre accessibili dall'esterno tutte le componenti definite nel modulo. Pertanto si potrà accedere alla rappresentazione di **complex** e scrivere

$z.R := x.I + y.R$

In questo modo si sta violando **il requisito di protezione:**
sui dati si deve poter operare solo con gli operatori definiti all'atto della specifica.

Requisito di protezione

Problema: **Violazione del requisito di protezione.**

Per soddisfare il requisito di protezione è indispensabile poter nascondere le implementazioni, distinguendo una **parte pubblica** del modulo da una **parte privata**.

La prima contiene tutto ciò che è esportabile mediante la clausola `uses` mentre la seconda contiene le componenti del modulo non visibili dall'esterno.

In particolare le **dichiarazioni** di identificatori di tipo e i prototipi* delle funzioni/procedure compaiono nella parte pubblica, mentre le **implementazioni** sono nella parte privata.

* prototipo della funzione = nome della funzione + il numero e tipo dei suoi parametri

```
module complessi;  
public  
  type complex;  
  function RealPart(complex):Real;  
  function ImmPart(complex):Real;  
  function ConsCx(Real,Real):Complex;
```

```
...
```

```
private  
  type complex= record  
                      R: real;  
                      I: real  
                    end;  
  function RealPart(c:complex):real  
  begin  
    return c.R  
  end;  
  function ImmPart(c:complex):real  
  begin  
    return c.I  
  end;  
  ...  
end module.
```

I Moduli: Parti Pubblica e Privata

La realizzazione del tipo *complex* e dei relativi operatori è ora nascosta all'utilizzatore del modulo. Quindi non sarà più lecito scrivere:

$z.R := x.I + y.R$

perché il programma che usa il modulo non sa che *complex* è di fatto realizzato come un record. La manipolazione di dati di tipo *complex* può avvenire solo attraverso gli operatori definiti nel modulo complessi. (*l'implementazione è privata*)

VANTAGGIO: garantire il requisito di protezione significa consentire i **controlli di consistenza dei tipi** sui dati ottenuti per astrazioni, oltre che sui dati primitivi per i quali l'esecutore del linguaggio assicura già tali controlli. **Quindi tutti i dati, siano essi primitivi o astratti, sono trattati in modo uniforme.**

Tipi astratti: definizione implicita dei valori

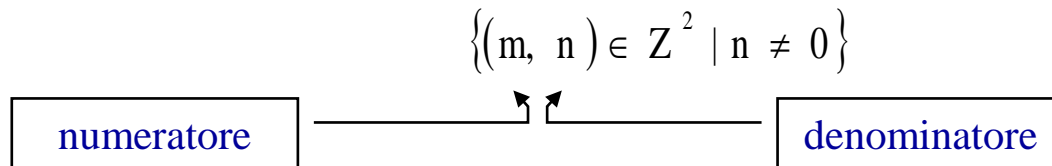
Si deve osservare che in questo modo non si è definito il tipo *complex* elencando esplicitamente tutti i valori del tipo. L'insieme dei valori del tipo astratto *complex* è definito solo **indirettamente**; esso consiste di tutti i valori che possono essere generati da successive applicazioni delle operazioni.

Esempio:

Si supponga di voler definire un tipo i cui valori sono numeri razionali, con operazioni aritmetiche (come l'addizione) che sono esatte.

Tipi astratti: definizione implicita dei valori

Formalmente si definisce l'insieme dei numeri razionali come



che fa pensare ad una rappresentazione mediante il prodotto cartesiano **Integer x Integer**. Purtroppo in questo modo non si possono escludere le coppie **(m,0)**, cioè con denominatore nullo. Insiemi di valori così complicati non possono essere definiti **direttamente** in nessun linguaggio di programmazione esistente. Se lo fossero, il type checking sarebbe estremamente complicato e ...

Tipi astratti: definizione implicita dei valori

... **dovrebbe essere dinamico**: infatti un controllo di tipo dovrebbe assicurare che il valore del denominatore non sia nullo e questo potrebbe essere fatto solo a run-time.

La soluzione consiste nel definire **implicitamente** i valori ammessi, attraverso la definizione di un tipo astratto **Rational**.

```
module razionali;  
public  
type Rational;  
const Rational zero;  
const Rational one;  
function ConsRat(Integer,Integer):Rational;  
function +(Rational,Rational):Rational;  
function =(Rational,Rational):Boolean;
```

```

private
  type Rational = record
    N: Integer;
    D: Integer
  end;
  const Rational zero=(0,1);
  const Rational one=(1,1);
  function ConsRat(num: Integer; den:Integer): Rational
  var razionale: Rational;
  begin
    if den <> 0 then
      begin
        razionale.N := num;
        razionale.D := den;
        return razionale
      end
    else
      writeln("Invalid rational number")
    end
  end
  ...
end module.

```

Tipi astratti: definizione implicita dei valori

Si osservi che il costruttore **ConsRat** costruisce solo numeri razionali leciti, cioè con denominatore non nullo. Inoltre gli altri operatori ammessi, come $+$, restituiscono solo numeri razionali leciti.

In questo modo è possibile specificare implicitamente il sottoinsieme di **Integer x Integer** che corrisponde a dei numeri razionali leciti.

Ma c'è di più. Si osservi che

$\frac{1}{2}$ $\frac{2}{4}$ $\frac{3}{6}$ $\frac{4}{8}$

sono tutti numeri razionali equivalenti. Una rappresentazione efficiente permetterebbe di rappresentare solo il più piccolo, che diventerebbe il rappresentante della classe di equivalenza.

Moduli: operatori privati

Finora tutte le operazioni definite nel modulo sono state considerate accessibili dall'esterno. Tuttavia, quando la realizzazione di tali operazioni risulta complessa, si potrebbe voler scrivere dei sottoprogrammi utilizzabili nella realizzazione delle operazioni ma non accessibili dall'esterno. Questi sono detti **privati** e compaiono solo nella parte privata del modulo.

```
module razionali;  
public  
type Rational;  
const Rational zero;  
const Rational one;  
function ConsRat(Integer,Integer):Rational;  
function +(Rational,Rational):Rational;  
function =(Rational,Rational):Boolean;
```

```

private
  type Rational = record
    N: Integer;
    D: Integer
  end;

  const Rational zero=(0,1);
  const Rational one=(1,1);
  function Prod(a,b,c,d: Integer):Integer;
  begin
    return ( a*d + b*c );
  end
...
  function +(r1,r2: Rational):Rational
  var r3: Rational;
  begin
    r3.N := prod(r1.N,r1.D,r2.N,r2.D);
    r3.D := r1.D*r2.D;
    return r3
  end
...
end module.

```

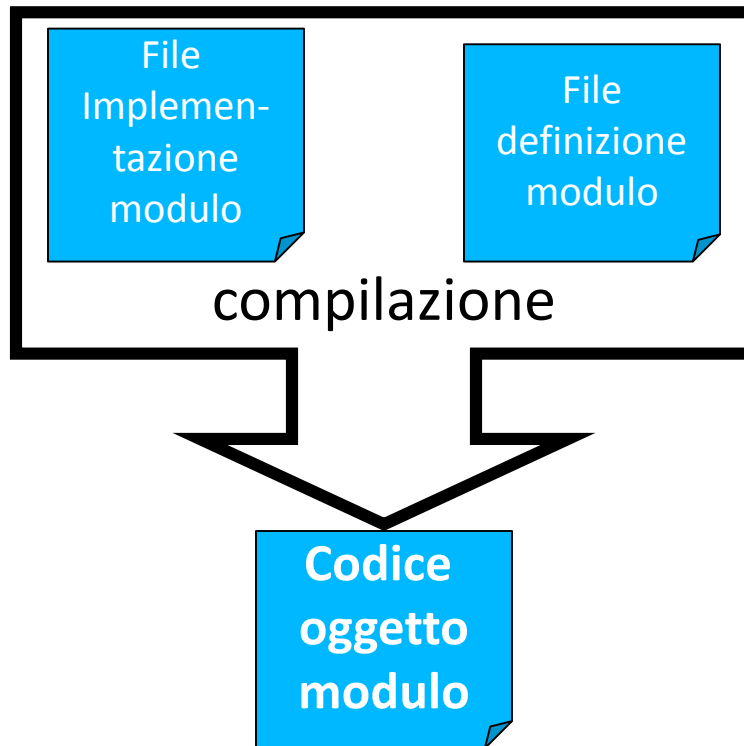
I Moduli: Distribuzione in file

Operativamente, ci si potrebbe aspettare che la parte pubblica di un modulo e quella privata finiscano in due file differenti (file di definizione e file di implementazione).

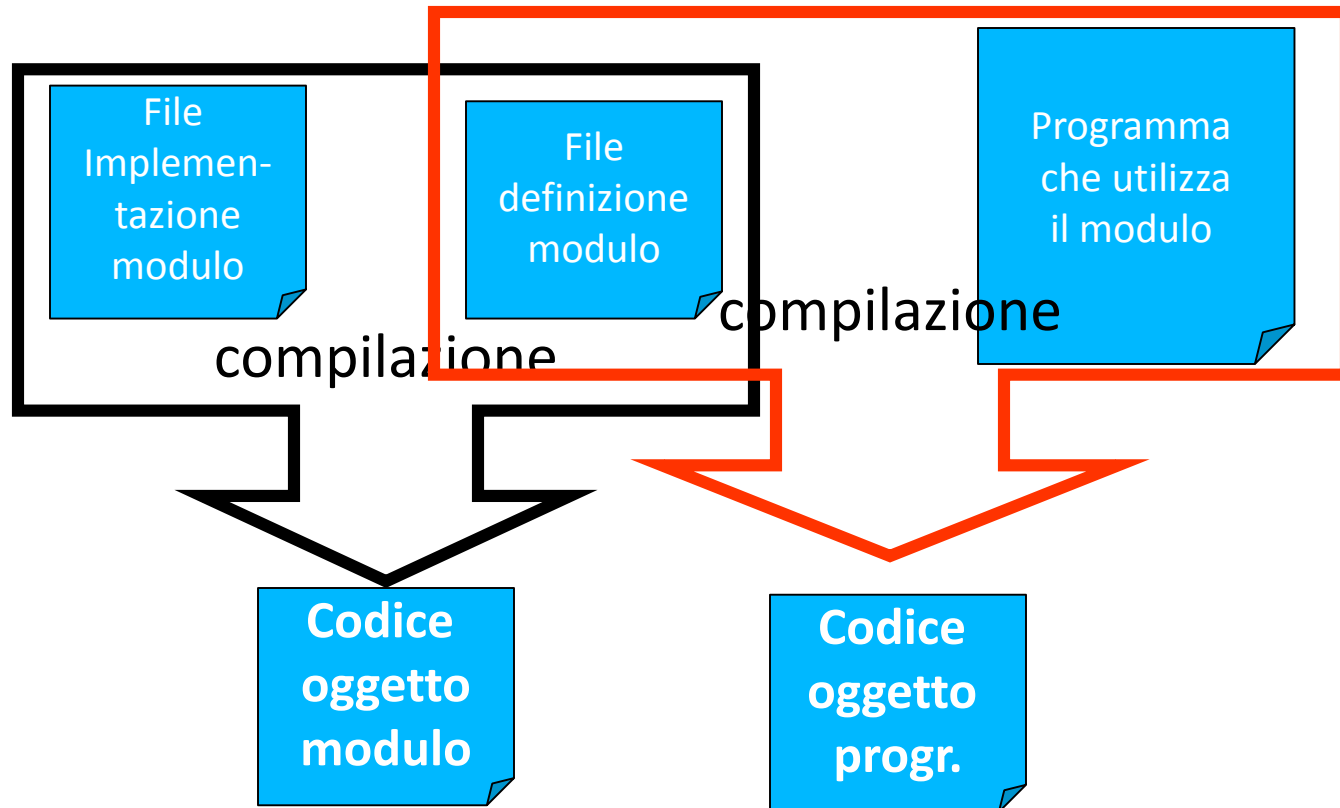
Il compilatore necessita di entrambi i file per produrre il codice oggetto del modulo. Tuttavia esso dovrebbe aver bisogno solo del file di definizione per produrre il codice oggetto di un programma che importa il modulo.

I due moduli oggetti dovrebbero poi essere collegati (fase di link) per produrre il file eseguibile.

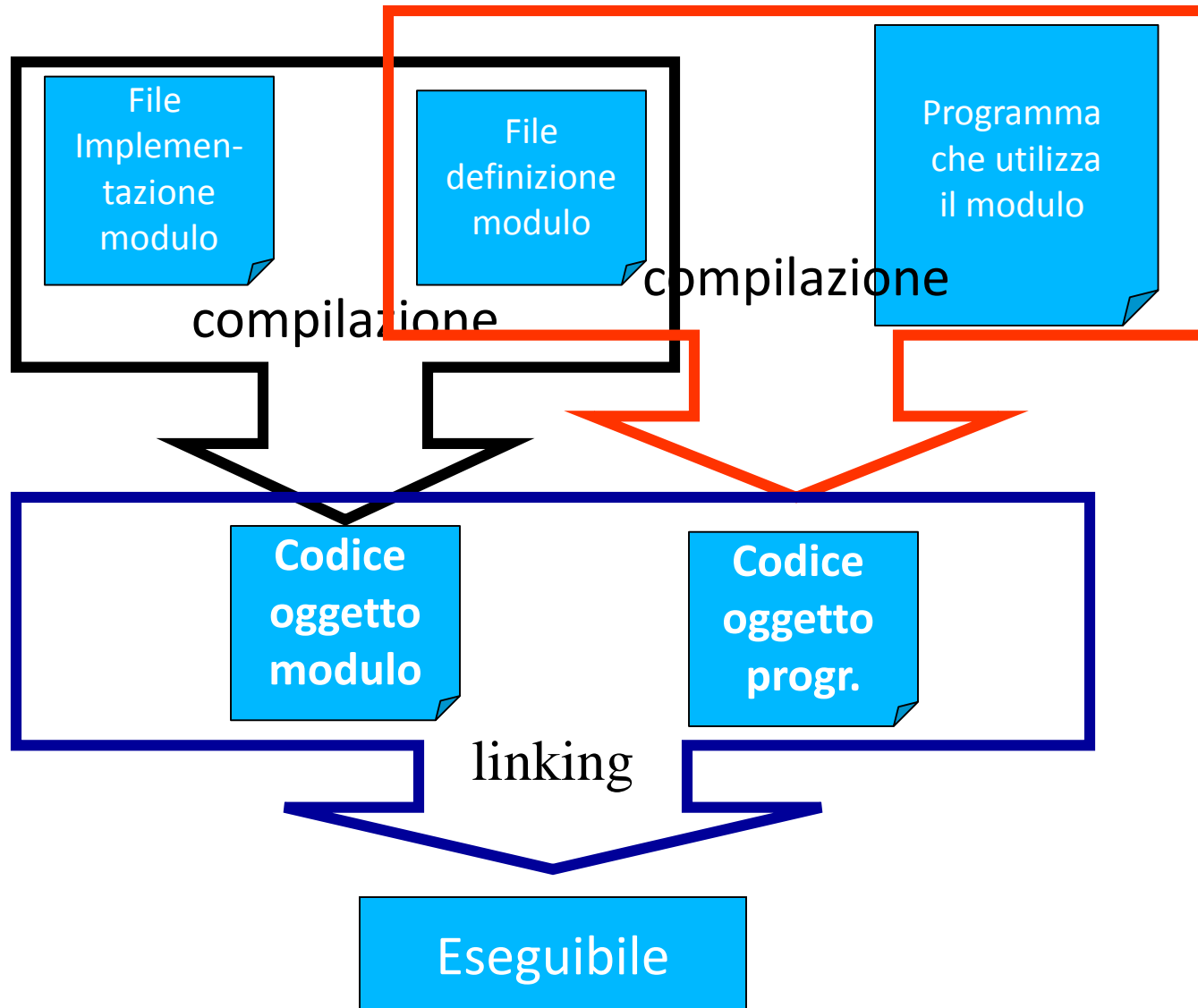
I Moduli: Distribuzione in file



I Moduli: Distribuzione in file



I Moduli: Distribuzione in file



I Moduli: Distribuzione in file

Vantaggio della separazione: è possibile distribuire il file di definizione insieme al codice oggetto del modulo senza fornire il file dell'implementazione.

Problema: consideriamo il modulo complessi e la seguente dichiarazione presente nel programma che lo importa:

`x,y,z: complex`

Essa non può essere compilata, perché non è nota la struttura di un dato di tipo complex e **non si sa quanto spazio di memoria riservare per le tre variabili.**

I Moduli: Distribuzione in file

Soluzioni:

1. Ricorrere ai tipi **opachi** del Modula-2, cioè a tipi quelli dichiarati nella parte pubblica ma definiti in quella privata, che devono essere obbligatoriamente **di tipo puntatore**. In questo modo il compilatore sa che ad ogni variabile dichiarata come tipo opaco dev'essere riservato lo spazio tipicamente riservato a un puntatore.

Quello che si nasconde effettivamente è la struttura dell'oggetto puntato, oltre alla implementazione degli operatori.

I Moduli: Distribuzione in file

Soluzioni:

2. Come accade nei package in Ada, il file di **definizione include sia le dichiarazioni pubbliche di tipi, costanti, funzioni e procedure e sia le definizioni private dei tipi e delle costanti**. Nel file di implementazione sono riportate solo le implementazioni delle procedure e delle funzioni. In questo modo l'informazione sulla dimensione di un dato *complex* è nota al compilatore. Tuttavia il compilatore non utilizzerà mai l'informazione sulla struttura per permettere l'accesso ai dati il cui tipo è dichiarato come privato.

I moduli con stato locale

I moduli che contengono solo definizioni di tipi, procedure e funzioni non hanno un proprio stato permanente (sono *stateless*). Le variabili definite nelle procedure e nelle funzioni del modulo hanno un tempo di vita pari a quello dell'esecuzione della procedura o funzione che le contiene.

Si può pensare di estendere la definizione di modulo, consentendo anche la *definizione di variabili esterne alle procedure e funzioni*. Tali variabili avranno un tempo di vita pari a quello dell'unità di programma che "usa" il modulo.

I moduli con stato locale

In questo modo il modulo viene dotato di un suo **stato locale**, che consiste nel valore delle variabili esterne a procedure e funzioni. Un modulo dotato di stato locale è chiamato **oggetto**.

Il termine **oggetto** è usato anche per la variabile nascosta del modulo.


Le funzioni e le procedure incapsulate in un oggetto sono comunemente chiamate **metodi**.

I moduli con stato locale

```
module stack10real;
public
    procedure push(real);
    function pop():real;
private
    type stack_object=
        record
            st:array[1..10] of real;
            top: 0..10;
        end;
    var s:stack_object;

    procedure push(elem:real)
    begin
        if s.top<10 then
            begin
                s.top:= s.top+1;
                s.st[s.top]:=elem;
            end;
        end;
    end;

    function pop():real;
    begin
        if s.top>0 then
            begin
                s.top := s.top-1;
                return s.st[s.top+1];
            end;
        end;
    end module.
```



A diagram consisting of a vertical line with an arrowhead at the top, pointing from the `end module.` line of the public `pop` function to the `function pop():real;` line of the private `pop` function, indicating a call to the private function.

I moduli con stato locale

Il tempo di vita della variabile *s* è quello del modulo.

La variabile *s* è usata ripetutamente in chiamate successive degli operatori *push* e *pop*. Il suo valore condiziona anche l'ordine di chiamata delle procedure, per cui non è possibile invocare la funzione *pop* se non dopo una *push* che ne modifica il campo *top*.

Problema: come facciamo ad assicurarci che *top* sia inizializzata precedentemente alla prima invocazione della procedura *push*?

I moduli con stato locale

Ovviamente **per top** basterebbe una dichiarazione con **inizializzazione**.

Più in generale, per poter inizializzare l'intera struttura **s** si potrebbero adottare due soluzioni:

1. Dotare il modulo di una procedura **init()** da invocare dopo aver importato il modulo.
2. Dotare il modulo di un *main*, eseguito solo nel momento in cui il modulo è importato in altri moduli mediante “uses”.

La seconda soluzione è preferibile perché non si basa sul corretto utilizzo del modulo da parte del programmatore.

I moduli con stato locale

```
module stack10real;
public
  procedure push(real);
function pop():real;
private
  type stack_object=
    record
      st:array[1..10] of real;
      top: 0..10;
    end;
  var s:stack_object;
  ...
procedure push(elem:real)
begin
  if s.top<10 then
    begin
      s.top:= s.top+1;
      s.st[s.top]:=elem;
    end;
end;
end;
```



```
function pop():real;
begin
  if s.top>0 then
    begin
      s.top := s.top-1;
      return s.st[s.top+1];
    end;
begin
  integer i;
  s.top:=0;
  for i:=1 to 10 s.st[i]:=0;
end;
end module.
```

I moduli con stato locale

Osservazioni:

1. La definizione del tipo **stack_object** è ora privata. L'utente può solo usare l'oggetto **stack10real** e non può dichiarare variabili di tipo **stack_object** nell'unità di programma che importa il modulo.
2. È possibile cambiare lo stato locale solo attraverso gli operatori **push** e **pop**, che costituiscono l'**interfaccia** dell'oggetto con il mondo esterno.
3. Il modulo può controllare l'ordine di accesso alla variabile nascosta, ad esempio, per impedire l'estrazione dei dati prima che la variabile venga avvalorata.

I moduli con stato locale

Quando una unità di programma **P** importerà il modulo **stack10real** con il comando:

uses stack10real

allora verranno allocati in memoria

- un array di 10 reali **st**
- una variabile intera **top**

che saranno comunque inaccessibili a **P**.

Inoltre verrà eseguito il main del modulo **stack10real** che provvederà a inizializzare **top** e gli elementi di **st** a zero.

I moduli con stato locale

Così le chiamate alla procedura

push(elem)

e alla funzione

pop()

modificheranno **top** e **st** senza che ciò sia osservabile dall'esterno.

I moduli con stato locale

Va osservato che in questo modo il programma **P** ha accesso ad un solo oggetto “stack di 10 valori reali”. Se ce ne fosse bisogno di un altro, potremmo scrivere un altro modulo, denominato **stack10real2**, analogo al precedente.

Tuttavia, in questo modo sorgerebbe il problema di come distinguere gli operatori push e pop, ora ambigui.

Soluzione: usare la notazione puntata per riferirsi al modulo appropriato sul quale invocare la procedura o la funzione.

I moduli con stato locale

Stack10real.push(elem)

Stack10real.pop()

Stack10real2.push(elem)

Stack10real2.pop()

Il Package in Ada

In Ada il modulo è chiamato **package** e si compone di due parti:

1. una **specification**, che interagisce con l'ambiente esterno: contiene le dichiarazioni di tipi, costanti, procedure e funzioni
2. un **body**, che contiene, fra l'altro, l'implementazione di procedure e funzioni dichiarate nella **specification**, ed eventualmente una routine di inizializzazione del package

Il Package in Ada

A sua volta la specification si articola in due sottoparti:

1. **visible**: le entità dichiarate in questa parte possono essere rese note ad altre unità di programma per mezzo della clausola use
2. **private**: le entità dichiarate in questa parte non possono essere né esportate e né dichiarate nel corpo

Il Package in Ada

La parte di specifica inizia con la parola chiave **package** seguita dall'identificatore del package e da **is**; seguono poi le *dichiarazioni* delle entità visibili e private.

Esempio:

```
package Type_complex is  
    type Complex is record  
        RL, IM: Real;  
    end record;  
    I: constant Complex := (0.0, 1.0)  
    function "+"(x,y: Complex) return Complex  
    ...  
end Type_complex;
```

Il Package in Ada

Questo è un esempio di specifica di un package che realizza il tipo astratto **Complex**. In questo non c'è una parte privata nella specifica, sicché un programma che fa uso di questo package può operare su variabili complesse mediante espressioni come la seguente:

```
C.IM := C.IM + 1.0;
```

invece di ricorrere alla forma più astratta:

```
C:= C + I;
```

Per evitare il problema possiamo nascondere la struttura del tipo **Complex** nella parte privata.

II Package in Ada

```
package Type_complex is  
    type Complex is private;  
    I: constant Complex;  
    function "+"(x,y: Complex) return Complex;  
    ...  
private  
    type Complex is record  
        RL, IM: Real;  
    end record;  
    I: constant Complex := (0.0, 1.0)  
end Type_complex;
```

N.B.: avendo dichiarato il tipo Complex nella parte privata non è più possibile inizializzare la costante I nella parte pubblica, in quanto non è ancora nota la rappresentazione di Complex.

Il Package in Ada

Anche il corpo del package incomincia con la parola **package** seguita però dalla parola **body**, dall'identificatore e da **is**. Nel corpo sono fornite le implementazioni delle procedure e funzioni dichiarate nella corrispondente specification.

```
package body Type_complex is  
function "+"(x,y: in Complex) return Complex is  
begin  
    return(x.RL+y.RL, x.IM+y.IM);  
end "+";  
...  
end Type_complex;
```

Il Package in Ada

Ovviamente la struttura del tipo `Complex` è visibile alla parte `body` del package, quindi si potrà accedere ai campi `RL` e `IM` dei record `Complex`.

Il package potrà essere utilizzato come segue:

```
with Type_complex; use Type_complex;
```

```
procedure main is
```

```
    cpx1, cpx2: Complex;
```

```
begin
```

```
    ...
```

```
    cpx1 := cpx2 + I;
```

```
    ...
```

```
end main;
```

Il Package in Ada

Si osservi che sia nella specifica e sia nel corpo di `Type_complex` non c'è alcuna dichiarazione di variabili esterne a procedure e funzioni. Ciò vuol dire che questo package non è dotato di uno stato locale, cioè *non definisce un oggetto*.

Per questa ragione il corpo del package non necessita di un “main”: non si deve inizializzare un oggetto.

Il package *può in ogni caso avere un proprio main*. Esso verrà specificato dopo le varie procedure e funzioni e sarà compreso fra un `begin` e l'`end` del package.

II Package in Ada

```
with Simple_io; use Simple_io;  
package body Type_complex is  
function "+"(x,y: in Complex) return Complex is  
begin  
    return(x.RL+y.RL, x.IM+y.IM);  
end "+";  
...  
begin  
    put("Main of package Type_complex ");  
end Type_complex;
```

L'esecuzione del main del package avverrà al momento in cui si importa il package mediante la clausola use. Si vedrà quindi visualizzare la frase "Main of package Type_complex"

II Package in Ada

```
package Type_stack is  
    type Stack is private  
    procedure push(s:in out Stack; x:in Integer);  
    procedure pop(s:in out Stack);  
    procedure top(s:in Stack; x:out Integer);  
    function empty(s:in Stack) return Boolean;  
private  
    max: constant:= 100;  
    type Stack is limited record  
        st:array(1..max) of Integer;  
        top: Integer range 0..max := 0;  
    end record;  
end Type_stack;
```

```
package body Type_stack is  
    procedure push(s:in out Stack; x:in Integer) is  
    begin  
        s.top := s.top+1;  
        s.st(s.top) := x;  
    end push;  
    procedure pop(s:in out Stack) is  
    begin  
        s.top := s.top - 1;  
    end pop;  
    procedure top(s:in Stack; x:out Integer) is  
    begin  
        x := s.st(s.top);  
    end top;  
    function empty(s:in Stack) return Boolean is  
    begin  
        return(s.top=0);  
    end empty;  
end Type_stack;
```

Il Package in Ada

- Le specifiche del tipo astratto Pila prevedono anche un costruttore *CreaPila*, che non ha un corrispondente in questa definizione del tipo astratto Stack.
- La ragione è che si stanno utilizzando costruttori impliciti forniti nel linguaggio Ada, come la dichiarazione di una variabile di tipo Stack.
- Tuttavia **se un costruttore dovesse essere parametrizzato** (come quello di *Conto Con Fido*) sarà necessario prevedere un metodo.

Il Package in Ada

In Ada, definendo un tipo come *private* è possibile applicare su istanze di quel tipo tutti i metodi definiti nella parte pubblica della specifica, ma anche effettuare **assegnazioni** e **confronti di (dis-)eguaglianza**.

Queste operazioni che il compilatore offre ‘gratuitamente’ per un tipo privato devono necessariamente essere definite in modo generale, indipendentemente da come il tipo è poi definito.

Quindi saranno implementate semplicemente **copiando o confrontando byte a byte** le aree di memoria riservate a due dati dello stesso tipo dichiarato come privato.

Il Package in Ada

Problemi:

- 1) queste tre operazioni (assegnazione $\textcolor{red}{:=}$, confronto per eguaglianza $\textcolor{red}{=}$, confronto per disequaglianza $\textcolor{red}{\neq}$) potrebbero non far parte della specifica di un dato astratto. Quindi potrebbe non essere corrette offrirle all'utilizzatore del tipo dichiarato come privato.
- 2) la semantica delle operazioni potrebbe essere diversa da quella stabilita dal compilatore.
Facciamo un esempio ...

Il Package in Ada

```
with Type_stack, use Type_stack,  
procedure main is  
    st1, st2: Stack;  
    cmp: Boolean;  
begin  
    push(st1, 1);  
    push(st2, 1);  
    push(st1, 2);  
    pop(st1);  
    cmp := st1 = st2  
end main;
```

Il valore di cmp è False. Infatti i corrispondenti campi top dei record st1 ed st2 sarebbero identici, mentre non avrebbero gli stessi valori i corrispondenti campi st. Eppure i due stack sarebbero identici secondo la specifica algebrica

Il Package in Ada

Il tipo **limited private**

Per evitare tutto ciò, è sufficiente dichiarare il tipo come **limited private**, che inibisce l'uso delle operazioni di assegnazione e confronto offerte per default dal compilatore.

```
package Type_stack is
```

```
    type Stack is limited private
```

```
    procedure push(s:in out Stack; x:in Integer);
```

```
    procedure pop(s:in out Stack);
```

```
    procedure top(s:in Stack; x:out Integer);
```

```
    function empty(s:in Stack) return Boolean;
```

```
...
```

Il Package in Ada

In Ada è anche possibile definire **oggetti**, cioè moduli dotati di **stato locale**.

Esempio: oggetto stack.

```
package Stack is  
    procedure push(x:in Integer);  
    procedure pop;  
    procedure top(x:out Integer);  
    function empty return Boolean;  
end Stack;
```

package body Stack is

max: **constant**:= 100;

type Table **is** array(1..max) **of** Integer;

st: Table;

top: Integer **range** 0..max := 0;

procedure push(x:in Integer) **is**

begin

top := top+1;

st(top) := x;

end push;

procedure pop **is**

begin

top := top - 1;

end pop;

procedure top(x:out Integer) **is**

begin

x := st(top);

end top;

function empty **return** Boolean **is**;

begin

return(top=0);

end empty;

end Stack;

Il Package in Ada

Si potrà quindi utilizzare l'oggetto Stack nel modo seguente:

```
with Stack; use Stack;  
procedure main is  
begin  
....  
    push(1);  
    push(2);  
    pop  
    if empty then push(1);  
...  
end main;
```

Classi di oggetti


In generale, un oggetto è un insieme di variabili interne ad un modulo e manipolabili esternamente solo mediante gli operatori (pubblici) definiti nel modulo stesso.

Da quanto visto finora, per poter definire più oggetti “simili” o “dello stesso tipo”, cioè con medesima rappresentazione e stesso insieme di operatori, si è costretti a definire tanti moduli quanti sono gli oggetti che si vogliono usare nel programma. Tutti questi moduli differiranno solo per l'identificatore del modulo (*identificatore dell'oggetto*).

Per evitare l'inconveniente di dover duplicare un modulo si può pensare di definire un *modulo generico* che identifica una *classe* di oggetti simili. I singoli oggetti sono poi ottenuti con il meccanismo della *istanziamento* della classe.

Classi di oggetti

```
generic module stack10real;
public
    procedure push(real);
    function pop():real;
private
    type stack_object=
        record
            st:array[1..10] of real;
            top: 0..10;
        end;
    var s:stack_object;
    ...
    procedure push(elem:real)
    begin
        if s.top<10 then
            begin
                s.top:= s.top+1;
                s.st[s.top]:=elem;
            end;
        end;
    end;
```



```
function pop():real;
begin
    if s.top>0 then
        begin
            s.top := s.top-
1;
            return s.st[s.top+1];
        end;
    end;
begin
    integer i;
    s.top:=0;
    for i:=1 to 10 s.st[i]:=0
end;
end module.
```

Classi di oggetti

Continuando a estendere un ipotetico linguaggio di programmazione, una istanziazione potrebbe essere ottenuta mediante un costrutto di questo genere:

```
st1 instantiation of stack10real;  
st2 instantiation of stack10real;
```

Le istanziazioni creano due oggetti di tipo stack denotati rispettivamente **st1** e **st2**. Al momento della istanziazione verranno allocati in memoria

- due array di 10 real
- due variabili intere

Inoltre verrà eseguito per ognuna delle istanze (st1 e st 2) il main del modulo generico

Classi di oggetti

Agli oggetti si potrà accedere mediante gli operatori push e pop, ma con l'accorgimento di far precedere il nome dell'operatore dall'identificatore dell'oggetto, altrimenti il compilatore non potrebbe risolvere correttamente i riferimenti per via della loro definizione in entrambi i moduli **st1** e **st2**.

Quindi scriveremo:

```
st1.push(10)  
      ↑  
m := st2.pop()  
      ↑
```

Generic Package in Ada

In Ada un package che specifica e implementa un singolo oggetto può essere facilmente trasformato in un *generic package*, che definisce una classe di oggetti, premettendo la parola **generic** alla dichiarazione del modulo.

```
generic  
package STACK  
    procedure PUSH(X:in INTEGER) ;
```

In questo modo si definisce solo una *matrice* degli oggetti da creare. Per ottenere i singoli oggetti dobbiamo **istanziare** il generic package:

```
package ST1 is new STACK  
package ST2 is new STACK
```

Astrazione della dichiarazione di modulo

- La precedente definizione di una **classe** corrisponde a una particolare forma di astrazione, quella della classe sintattica '**dichiarazione di un modulo**'.
- L'operazione di istanziiazione corrisponde alla invocazione di questa astrazione ed ha l'effetto di 'creare legami' (binding). In particolare, si crea un legame fra
 - Identificatore dell'oggetto
 - Nome della classe (cioè nome del modulo generico)
- Pertanto la definizione di una classe corrisponde a una particolare forma di **astrazione generica**, cioè di astrazione applicata alla classe sintattica dichiarazione.

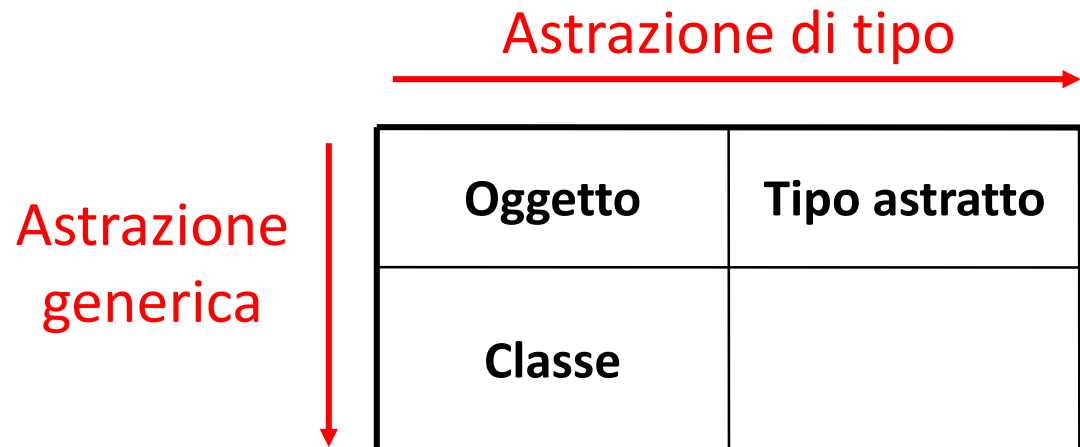
Astrazione della dichiarazione di modulo

- Quando si affronterà l'argomento dell'astrazione generica in modo sistematico, si osserverà che è possibile astrarre anche su altre dichiarazioni (per esempio, funzioni e procedure), oltre quella di modulo.

Tipi astratti o classi?

Riepilogando, se in fase di progettazione si identifica l'esigenza di disporre di un dato astratto, in fase realizzativa si può:

- 1) **Definire un oggetto**: la scelta è appropriata nel caso in cui si necessita di *una sola occorrenza* del dato astratto;
- 2) **Definire un tipo astratto**: l'astrazione riguarda la classe sintattica *tipo*;
- 3) **Definire una classe**: l'astrazione riguarda la dichiarazione di un modulo (*dotato di stato locale*).



Tipi astratti o classi?

In tutti i casi la rappresentazione del dato astratto viene nascosta e la manipolazione dei valori è resa possibile solo mediante operazioni fornite allo scopo

Tuttavia ci sono delle **differenze** fra tipo astratto e classe di oggetti ...

Tipi astratti o classi?

- *Sintattica*: nel tipo astratto gli operatori hanno **un parametro in più**, relativo proprio al tipo che si sta definendo.

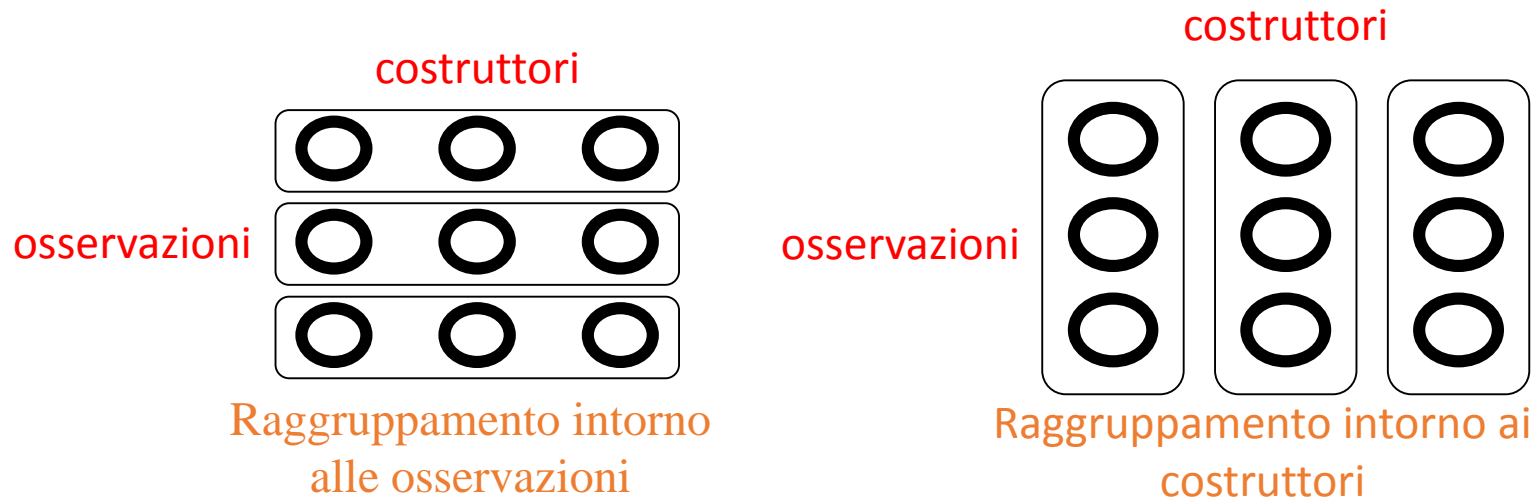
Esempio: nel tipo astratto Stack, gli operatori POP e PUSH hanno un parametro in più di tipo Stack che gli operatori definiti per la classe Stack non hanno.

- *Realizzativa*: nel caso del tipo astratto gli operatori sono definiti una sola volta, mentre nel caso della classe gli operatori sono definiti più volte, tante quante sono le istanze. Le diverse copie degli operatori agiranno su diversi dati (gli oggetti) in memoria centrale (*poiché ogni istanza avrà un suo stato*)

Tipi astratti o classi?

- **Concettuale**: richiamando la suddivisione delle operazioni su un dato astratto in osservazioni e costruttori (vedi specifiche algebriche), si può dire che:
 - il tipo astratto è **organizzato intorno alle osservazioni**. Ogni osservazione è implementata come una operazione su una rappresentazione concreta derivata dai costruttori. Anche i costruttori sono implementati come operazioni che creano valori. La rappresentazione è **condivisa** dalle operazioni, ma è nascosta ai fruitori del tipo astratto.
 - La classe è **organizzata intorno ai costruttori**. Le osservazioni diventano metodi dei valori. Un oggetto è definito dalla combinazione di tutte le osservazioni possibili su di esso.

Tipi astratti o classi?



In breve, un tipo astratto di dato può essere inteso **come un insieme con operazioni** (come un'algebra, insomma) mentre le classi sono **insiemi di operazioni**.

Vantaggi del tipo astratto

- Nei linguaggi imperativi, i valori di un tipo astratto vengono trattati alla stregua dei valori di un tipo concreto, cioè sono cittadini di **prima classe**. Al contrario i valori rappresentati mediante oggetti sono trattati come cittadini di **terza classe** in quanto:
 - una procedura non può restituire l'istanza di un generic package;
 - non è possibile creare dinamicamente degli oggetti (le istanze sono stabilite al momento della compilazione).
- I tipi astratti sono utili in tutti i paradigmi di programmazione, mentre gli oggetti, essendo variabili aggiornabili, si adattano bene solo a un paradigma di programmazione side-effecting
- La notazione usata per chiamare un'operazione di un tipo astratto è più naturale perché valori e variabili del tipo astratto sono argomenti espliciti

Svantaggi del tipo astratto

- *Scarsa estendibilità*: l'aggiunta di un nuovo costruttore comporta dei cambiamenti intrusivi nelle implementazioni esistenti degli operatori.

Ogni operatore dovrà essere opportunamente rivisto in modo da prevedere il trattamento di rappresentazioni ottenute con nuovi costruttori.

Esempio: si vuole implementare il dato astratto *geometricShape* la cui specifica algebrica è fornita di seguito:

<i>osservazioni</i>	<i>Costruttore di g</i>	
	<i>square(x)</i>	<i>circle(r)</i>
<i>area(g)</i>	x^2	πr^2

Svantaggi del tipo astratto

Esempio (cont.): Definendo un **tipo astratto** *geometricShape* occorrerà scrivere un modulo che definisce tale tipo e include tre funzioni:

<i>square(x: real): geometricShape</i>	crea un quadrato di lato x;
<i>circle(r: real): geometricShape</i>	crea un cerchio di raggio r;
<i>area(g: geometricShape): real</i>	calcola l'area

Infatti, **tutte le operazioni che hanno necessità di operare sulla rappresentazione di *geometricShape* devono stare nel modulo dove essa è definita (raggruppamento intorno alle operazioni).**

Una possibile rappresentazione in Pascal per *geometricShape* sarà la seguente:

```
type geometricShape = record
  shape: char;
  value: real
end
```

Una possibile realizzazione dei tre operatori è:

Svantaggi del tipo astratto

Esempio (cont.):

```
function square(x: real):  
    geometricShape
```

```
var g: geometricShape
```

```
begin
```

```
    g.shape := 's';
```

```
    g.value := x;
```

```
    return g
```

```
end
```

```
function area(g: geometricShape): real
```

```
begin
```

```
    if g.shape = 's' then return g.value * g.value
```

```
    else return 3.14 * g.value * g.value;
```

```
end
```

```
function    circle(x:    real):  
    geometricShape
```

```
var g: geometricShape
```

```
begin
```

```
    g.shape := 'c';
```

```
    g.value := x;
```

```
    return g
```

```
end
```

Svantaggi del tipo astratto

Realizzando il dato astratto mediante **classi** si possono definire due moduli generici, uno per ogni forma geometrica (**o costruttore**).

```
generic module Circle
public
  function area(): real;
  procedure init(real);
private
  var raggio:real;
  procedure init(x:real)
  begin
    raggio := x
  end;
  function area():real
  begin
    return 3.14*raggio*raggio
  end;
end module.
```

```
generic module Square
public
  function area(): real;
  procedure init(real);
private
  var lato:real;
  procedure init(x:real)
  begin
    lato := x
  end;
  function area():real
  begin
    return lato*lato
  end;
end module.
```

Per utilizzare una forma geometrica si deve istanziare una delle due classi e invocare il metodo *init*.

Svantaggi del tipo astratto

Cosa succede se si estende la specifica del dato astratto in modo da considerare anche i rettangoli?

<i>osservazioni</i>	<i>Costruttore di g</i>		
	<i>square(x)</i>	<i>circle(r)</i>	<i>rectangle(l,m)</i>
<i>area(g)</i>	x^2	πr^2	$l \cdot m$

Se abbiamo specificato un **tipo astratto**, siamo stati costretti a cambiare la rappresentazione in modo da memorizzare due valori (per i due lati del rettangolo) e non uno.

```
type geometricShape = record
  shape: char;
  value: real;
  value2: real
end
```

Svantaggi del tipo astratto

Inoltre dobbiamo aggiungere l'opportuno costruttore ...

```
function rectangle(x,y: real):  
    geometricShape
```

```
var g: geometricShape  
begin
```

```
    g.shape := 'r';
```

```
    g.value := x;
```

```
    g.value2 := y;
```

```
    return g
```

```
end
```

... e dobbiamo modificare la
anche la funzione area ☹

```
function area(g: geometricShape): real
```

```
begin
```

```
if g.shape = 's' then return g.value * g.value
```

```
    else if g.shape='c' then return 3.14 * g.value  
        * g.value
```

```
        else return g.value * g.value2
```

```
end
```

Svantaggi del tipo astratto

Diversamente, avendo realizzato il dato astratto mediante **classi** basta aggiungere un'altra classe.

```
generic module Rectangle
public
  function area(): real;
  procedure init(real,real);
private
  var base, altezza:real;
  procedure init(x,y:real)
  begin
    base := x;
    altezza := y
  end;
  function area():real
  begin
    return base*altezza
  end;
end module.
```

Per utilizzare un rettangolo si dovrà istanziare questa classe e invocare il metodo *init*.

Nella programmazione orientata a oggetti, per evitare la riscrittura di *codice comune* ad altre classi già definite è possibile ricorrere ai meccanismi di **ereditarietà** tra le classi.

Astrazione generica

- Il concetto di classe di oggetti va inquadrato in un tema più generale, quello dell'**astrazione generica**.
- Il principio di astrazione suggerisce che si può astrarre anche sulla classe sintattica **dichiarazione**, in quanto essa sottintende una computazione. In particolare la valutazione di una dichiarazione comporta la creazione di **legami** (**bindings**).
- Un'**astrazione generica** è un'astrazione su una dichiarazione, pertanto il corpo della dichiarazione di una astrazione generica è a sua volta una dichiarazione. La chiamata di un'astrazione generica è detta **istanziamento** e produce dei legami elaborando la dichiarazione contenuta nel corpo dell'astrazione generica.

Astrazione generica

Analogamente a quanto detto per le altre astrazioni, l'astrazione generica potrà essere specificata come segue:

generic $I(FP_1; \dots; FP_n)$ is D

dove **I** è un identificatore dell'astrazione generica, **$FP_1; \dots; FP_n$** sono parametri formali, e **D** è una dichiarazione che, quando elaborata, produrrà dei legami. **D** funge da *matrice* dalla quale ricavare le dichiarazioni per istanziazione.

Una dichiarazione D può essere:

- La dichiarazione di un tipo;
- La dichiarazione di un modulo;
- La dichiarazione di una funzione;
- La dichiarazione di una procedura;
- ...

Astrazione generica

Occorre pensare a dei meccanismi per poter distinguere le diverse dichiarazioni che si ottengono per istanziazione. Un modo è quello di specificare un diverso identificatore all'atto della istanziazione, come nel seguente esempio:

A instantiation of I;

Così, la seguente dichiarazione generica

```
generic typeRxR is  
  type RxR = Record  
    x: real;  
    y: real  
end;
```

potrà essere utilizzata come matrice per generare le dichiarazioni per i tipi *Point2D* e *Complex*:

Point2D instantiation of typeRxR

Complex instantiation of typeRxR

Astrazione generica

Si è visto un utilizzo dell'astrazione generica nella dichiarazione di moduli dotati di stato locale.

Mediante l'operazione di istanziamento si ottengono diverse copie dell'oggetto che differiscono solo per il nome dell'identificatore.

L'astrazione generica di un oggetto corrisponde al concetto di classe.

Generic Package in Ada

Il *generic package* di Ada è una esemplificazione di astrazione generica. L'espressione:

```
package ST1 is new STACK
```

è un esempio di istanziazione generica.

Astrazione generica

Le astrazioni generiche, come qualsiasi altra astrazione, possono essere parametrizzate.

Esempio: Nel seguente esempio, viene definita una classe coda in Ada. La variabile **items** è un array di caratteri. Al fine di svincolare la definizione di classe da particolari costanti legate all'applicazione si dota l'astrazione generica del *parametro formale*, **capacity**, che è utilizzato per dimensionare l'array. L'istanziamento deve consentire di specificare il parametro effettivo, che sarà un valore da associare al parametro formale. In Ada l'istanziamento sarà specificata come segue:

```
package line_buffer is new queue_class(120) ;
```

Astrazione generica

generic

capacity: Positive;

package queue_class is

 procedure append(newitem: in Character);

 procedure remove(olditem: out Character);

end queue_class;

package body queue_class is

 items: array(1..**capacity**) of Character;

 size, front, rear: Integer range 0..**capacity**;

 procedure append(newitem: in Character) is

 ... ;

 procedure remove(olditem: out Character) is

 ... ;

begin

...

end queue_class;

Astrazione generica

In principio si può applicare l'astrazione generica a qualunque dichiarazione, incluso le procedure e le funzioni. Ad esempio, si potrebbe dichiarare una procedura **T_swap** per scambiare dati di un tipo **T** **predefinito**:

generic

```
procedure T_swap(a,b: in out T);  
procedure T_swap(a,b: in out T) is  
  tmp: T;
```

begin

```
    tmp :=a; a:=b; b:=tmp;
```

end T_swap;

e ottenere diverse copie di essa per istanziiazione:

```
procedure swap1 is new T_swap
```

```
procedure swap2 is new T_swap
```

Astrazione generica

In realtà è poco utile disporre di due funzioni identiche ma di nome diverso. Diversa sarebbe la situazione se potessimo dichiarare una generica procedure **T_swap** che opera su dati di tipo T qualunque, e potessimo specificare il tipo al momento dell'istanziamento. Per ottenere questo risultato necessitiamo di una particolare classe di **parametri**, quelli **di tipo**.

```
generic
  type T is private;
  procedure T_swap(a,b: in out T);
  procedure T_swap(a,b: in out T) is
    tmp: T;
begin
    tmp :=a; a:=b; b:=tmp;
end T_swap;
```

Astrazione generica

La clausola *generic* introduce un **parametro di tipo** e la dichiarazione che segue introduce la matrice di una procedura che scambia due dati di un tipo **T** generico. Le procedure effettive sono ottenute istanziando la procedura generica con i parametri di tipo effettivi da sostituire a **T**.

```
procedure int_swap is new T_swap(INTEGER) ;  
procedure str_swap is new T_swap(STRING) ;
```

Assumendo che *i* e *j* sono variabili di tipo **INTEGER** e che *s* e *t* sono variabili di tipo **STRING** allora:

```
int_swap(i, j) ;  
str_swap(s, t) ;
```

```
int_swap(i, s) ;  
str_swap(s, j) ;  
str_swap(i, j) ;
```

Astrazione generica

In questo modo si è:

- Svincolato la definizione dello scambio di due elementi da un fattore marginale, come il tipo degli elementi da scambiare
- Garantito comunque il **controllo statico dei tipi** fra parametri formali e parametri effettivi delle diverse procedure ottenute, e fra sorgente e destinazione di una assegnazione

L'uso dei parametri di tipo in astrazioni generiche offre un buon **compromesso** fra necessità di dover effettuare il **controllo statico dei tipi** e **desiderio di definire componenti software riutilizzabili**

Astrazione generica

I parametri di tipo possono essere utilizzati anche quando si definiscono delle classi, come indicato in questo esempio (in Ada)

generic

max: Positive;

type ITEM **is private**;

package Stack **is**

procedure push(x:in ITEM);

procedure pop;

procedure top(x:out ITEM);

function empty **return** Boolean;

end Stack;

package body Stack **is**

type Table **is array**(1..max) **of** ITEM;

st: Table;

top: Integer **range** 0..max := 0;

Astrazione generica

procedure push(x:in **ITEM**) **is**

begin

 top := top+1;

 st(top) := x;

end push;

procedure pop **is**

begin

 top := top - 1;

end pop;

procedure top(x:out **ITEM**) **is**

begin

 x := st(top);

end top;

function empty **return** Boolean **is**

begin

...

Astrazione generica

In questo caso per creare i singoli oggetti scriveremo:

declare

```
package STACK_INT is new STACK(10, INTEGER);  
use STACK_INT;  
package STACK_REAL is new STACK(10, REAL);  
use STACK_REAL;  
A: REAL; B: INTEGER;
```

begin

```
push(12);push(15.0);top(B);top(A)
```

end

Si osservi che non è necessario utilizzare la notazione puntata:

```
STACK_INT.push(10)
```

```
STACK_REAL.push(15.0)
```

in quanto push e top sono differenziate dal contesto (tipo di parametro effettivo passato).

Questo è un caso di *overloading* come si chiarirà meglio in seguito.

Astrazione generica

L'astrazione generica è quindi di supporto all'astrazione dati, in quanto permette di definire delle classi *che sono invarianti ad alcuni tipi di dati necessari per definirle*

Non solo. L'astrazione generica, mediante i parametri di tipo, è applicabile a tipi astratti che possono essere così ugualmente *svincolati dalla necessità di specificare il tipo degli elementi sui quali operare*

Esempio:

Si consideri il problema di definire dei tipi astratti per una applicazione che usa:

- 1) Stack di interi;
- 2) Stack di reali;
- 3) Stack di un tipo astratto *point3D* utilizzato per rappresentare i punti di uno spazio tridimensionale.

Cosa fare?

Astrazione generica

Una alternativa sarebbe quella di scrivere una definizione separata per ciascuno dei tre tipi.

Svantaggi:

1. **Codice duplicato** in quanto plausibilmente simile per tutte le definizioni (differisce solo nelle parti in cui si fa riferimento ai singoli elementi dello stack)
2. **Sforzo di programmazione ridondante**
3. **Manutenzione complicata** poiché le modifiche, come l'aggiunta di un nuovo operatore, vanno plausibilmente effettuate in tutte le versioni

Una alternativa sarebbe quella di *separare le proprietà di uno stack dalle proprietà dei loro elementi*.

Come? Utilizzando i parametri di tipo.

Astrazione generica

generic

MAX: Positive;

type ITEM is private;

package STACKS is

 type STACK is limited private;

 procedure PUSH(S:in out STACK; E:in ITEM);

 procedure POP(S: in out STACK; E: out ITEM);

private

 type STACK is

 record

 ST: array(1..MAX) of ITEM;

 TOP: integer range 0..MAX;

 end record;

end;

Astrazione generica

```
package body STACKS is
    procedure PUSH(S: in out STACK; E: in ITEM);
begin
    S.TOP := S.TOP - 1;
    S.ST(S.TOP) := E;
end PUSH;
procedure POP(S: in out STACK; E: out ITEM);
begin
    E := S.ST(S.TOP);
    S.TOP := S.TOP - 1;
end POP;
end STACKS
```

In questo modo si è definito un ***tipo astratto generico*** STACK.

Astrazione generica

I diversi stack richiesti dall'applicazione sono ottenuti per istanziazione:

```
declare
    package MY_STACK is new STACKS(100, REAL);
    use MY_STACK;
    x: STACK; I: REAL;
begin
    push(x, 175.0);
    pop(x, I)
end
```

Astrazione generica

Quando si parametrizza rispetto al tipo come
nell'esempio di **T_swap** dove avevamo le seguenti
assegnazioni:

tmp := a; a := b; b := tmp;

chi garantisce che l'assegnazione sia una operazione
valida per i dati di tipo **T**?

Poiché abbiamo astratto sul tipo non sappiamo se quel
tipo supporta l'assegnazione

Astrazione generica

In Ada l'espressione

`type T is private;`

sottintende che l'assegnazione e i predicati $=$ e \neq sono operazioni valide per il tipo effettivo denotato da T. Per questo, tutti gli esempi visti precedentemente non creavano problemi.

Se T fosse stato definito come `limited private` avremmo avuto problemi. Perché?

Supponiamo ora di voler definire un package di funzioni matematiche.

Le tre operazioni di assegnazione e (dis-)eguaglianza non saranno più sufficienti. Il parametro di tipo dovrà disporre di operazioni aritmetiche.

generic

type REAL is digits <>;

package numerical is

function sqrt(x:in REAL) return REAL;

...

end numerical;

package body numerical is

function sqrt(x:in REAL) return REAL is

root: REAL := x/2.0;

begin

while (abs (x-root2) > 2.0 * x) loop**

root := (root + x/root) / 2.0;

end loop;

return root;

end;

...

end numerical;

L'espressione

type REAL is digits <>;

stabilisce che il tipo effettivo denotato con REAL dev'essere un floating point, cioè un FLOAT o un LONG-FLOAT. All'interno del corpo del package possiamo usare gli operatori >, +, *, /, **, oltre a :=, = e /=.

Astrazioni generica

Il package può essere istanziato fissando come argomento un qualunque tipo *floating point*:

```
package single_precision is new numerical (Float);  
package double_precision is new numerical (Long_Float);  
...  
a, b: Float;  
...  
... single_precision.sqrt(a**2+b**2) ....
```

Ad ogni istanziazione di **numerical** il compilatore controlla che il parametro effettivo sia un tipo floating-point.

Astrazione generica vincolata

Più in generale, in un'astrazione generica si può specificare un **vincolo** su un parametro di tipo. In Ada tale specifica segue la seguente sintassi:

type T is *specifica delle operazioni applicabili al tipo T;*

Il compilatore controlla ciascuna istanziazione generica per assicurarsi che:

operazioni applicabili al tipo argomento \supseteq operazioni specificate come applicabili a T

e controlla la stessa astrazione generica per assicurarsi che:

operazioni specificate come applicabili a T \supseteq operazioni usate per T nell'astrazione generica

Insieme queste proprietà garantiscono che ogni operazione effettuata su dati di tipo T nell'astrazione generica sia di fatto applicabile al tipo argomento.

Astrazione generica vincolata

Esempio:

generic

type T **is** (<>) ;

procedure ord_2(a,b: in out T) ;

procedure ord_2(a,b:in out T) **is**

 tmp: T;

begin

if a > b **then**

 tmp :=a; a:=b; b:=tmp;

end if

end ord_2;

Nell'esempio c'è un singolo tipo T che dev'essere **discreto** (vincolo (<>)). L'istanziamento:

procedure order is new ord_2(NATURAL);

definisce la procedura order in grado di operare su numeri naturali.

Astrazione generica vincolata

Continuando, si supponga di voler definire una funzione generica che determina il minimo fra due valori (non necessariamente discreti). Potremmo scrivere qualcosa del genere:

generic

```
type T is private;
```

```
function minimum(x,y: in T) return T;
```

```
function minimum(x,y: in T) return T is
```

```
begin
```

```
    if x<=y then return x;
```

```
    else return y
```

```
end minimum;
```

Purtroppo questa dichiarazione generica non passa il vaglio del compilatore, perché tutto ciò che si garantisce per il tipo T è che esso dispone dei tre operatori di assegnazione e (dis-)eguaglianza. Nulla è stabilito per quanto riguarda l'operatore **<=**.

Astrazione generica vincolata

Tuttavia la seguente dichiarazione è legale:

```
generic
```

```
    type T is private;
```

```
    with function "<=" (a,b: in T) return BOOLEAN is <>;
```

```
function minimum(x,y: in T) return T;
```

```
function minimum(x,y: in T) return T is
```

```
begin
```

```
    if x<=y then return x;
```

```
    else return y
```

```
end minimum;
```

La parola chiave **with** è utilizzata per introdurre un parametro formale di tipo sottoprogramma (procedura o funzione). Ora, se **STACK** è il tipo effettivo per il quale è definita una funzione:

```
function T1_le (a,b: in STACK) return BOOLEAN;
```

la dichiarazione generica può essere istanziata così:

```
function T1_minimum is new minimum(STACK, T1_le);
```

Astrazione generica vincolata

D'altro canto se sul dato di tipo STACK fosse definita una funzione

```
function "<=" (a,b:in STACK) return BOOLEAN;
```

si potrebbe omettere il secondo parametro dalla istanziatura generica. Ad esempio:

```
function T1_minimum is new minimum(STACK) ;
```

Ciò è possibile perché si è specificato <> nel corpo della funzione parametro "<=".

Astrazione generica in C++

In C++ l'astrazione generica è supportata mediante i *template*.

Un template è del codice generico dotato di parametri che possono assumere valori specifici al momento della compilazione (compile-time).

Ad esempio, piuttosto che scrivere due classi, `ListOfInts` e `ListOfStrings`, si potrebbe scrivere una singola classe template:

```
template<class T> class List
```

dove il parametro di tipo (*class*) `T` del template può essere rimpiazzato da un qualunque tipo quando il codice è compilato.

Ciò è ottenuto mediante una istanziazione del template:

```
List<int> l1;
```

```
List<string> h2;
```

```
List<int> l3;
```


Astrazione generica in C++

```
template<class T> class inutile {  
    T x;  
public:  
    T getx() {return x;}  
    void setx(T y) { x = y};  
};
```

Possiamo creare due istanziazioni della classe template, usando in questo caso l'istanziamento esplicito:

```
template class inutile<int>  
template class inutile<char>
```

Il compilatore genererà due definizioni di classi, una per ogni istanziazione.

Astrazione generica in C++

```
class inutile<int> {  
    int x;  
public:  
    int getx() {return bar;}  
    void setx(int y) { x = y};  
};  
class inutile<char> {  
    char x;  
public:  
    char getx() {return bar;}  
    void setx(char y) { x = y};  
};
```

In generale ogni istanziazione di un template produce una copia del codice template.

La copia è creata in fase di precompilazione.

Astrazione generica in C++

Si distinguono due categorie di template in C++:

- 1. *Template di classe***: definisce la struttura e le operazioni per un insieme illimitato di tipi correlati.

Esempio:

Un singolo template di classe `Stack` potrebbe fornire una definizione comune per una pila di `int`, di `float`, e così via. Nella seguente dichiarazione della classe `Stack` il parametro di tipo `T` denota il tipo degli elementi contenuti in una pila.

Astrazione generica in C++

```
template<class T> class Stack{
    int top;
    int size;
    T* elements;
public:
    Stack(int n) {size=n; elements=new T(size); top =0;}
    ~Stack() {delete[] elements;}
    void push(T a)      {top++; elements[top]=a;}
    T pop()   {top--; return elements[top+1];}
};
```

Quando Stack è usato come nome di tipo, esso dev'essere accompagnato da un tipo come parametro esplicito.

Stack<int> s(99); ← stack di interi di dim. 99

Stack<char> t(80); ← stack di caratteri di dim. 80

Astrazione generica in C++

2. Template di funzione: definisce una famiglia di funzioni, la logica della funzione può essere utilizzata su diversi tipi di dati

Esempio:

Una famiglia di funzioni di ordinamento potrebbe essere dichiarata in questo modo:

```
template<class T> void sort (vector<T>);
```

Una funzione generata da una template di funzione è chiamata **funzione template**.

```
vector <complex> cv(100);  
vector <int> ci(200);  
void f(vector <complex> &cv, vector<int> &ci)  
{  
    sort(cv);  
    sort(ci);  
    sort(cv);  
}
```

Astrazione generica in C++

Si osservi che in C++ **non è necessaria l'istanziazione esplicita** dei template.

Nell'esempio, saranno istanziate due funzioni:

`sort(vector<complex>)` e `sort(vector<int>)`

Le diverse **istanze sono generate sulla base dei tipi dei parametri effettivi** delle chiamate a funzioni generiche.

Come si vede c'è ambiguità nella invocazione della funzione *sort*. In particolare, `sort(cv)` si riferisce a *sort(vector<complex>)* mentre `sort(ci)` si riferisce a *sort(vector<int>)*.

Si ha un caso di **overloading**, ovvero di associazione dello stesso identificatore di funzione a realizzazioni differenti. Il compilatore risolve questa ambiguità sulla base del tipo degli argomenti effettivi.

Generics in Java

Alcuni linguaggi di programmazione orientati a oggetti, come Java e C#, sono caratterizzati da:

- **Un sistema di tipi unificato**: tutte le classi ereditano da un'unica classe base
 - Es. la classe Object in Java
- **Riflessione**: consente di scoprire al run-time l'informazione su un oggetto, e in particolare, la sua classe (*run-time type identification*).

Questo facilita la scrittura di codice generale, senza dover ricorrere all'astrazione generica.

Generics in Java

Consideriamo la classe Java `ArrayList`. Essa dispone di due metodi:

`boolean add(Object o)` ← inserisce un oggetto in coda

`Object get(int index)` ← restituisce l'oggetto memorizzato all'indice `index`.

Se **a** è una variabile di tipo `ArrayList` e **s** è una variabile di tipo `stringa`, allora possiamo scrivere:

```
a.add(s);
```

```
s= (String) a.get(index);
```

dove `(String)` fa il cast dell'oggetto restituito (di classe `Object`) in una stringa.

Generics in Java

Un programmatore potrebbe aggiungere ad **a** un oggetto di tipo diverso da una stringa. In questo caso l'esecuzione di:

```
s = (String) a.get(index);
```

solleva un'eccezione (`ClassCastException`) che occorre gestire.

Una soluzione al problema è offerta dalle **Generics**, la cui sintassi è molto simile a quella dei template, ma è semanticamente molto diversa.

In questo caso si utilizza la classe `ArrayList<E>` dove il parametro **E** si riferisce alla classe di oggetti che occorre memorizzare.

*N.B.: In Java, a differenza del C++, non è necessario specificare che **E** è una classe.*

Generics in Java

Come accade per i template C++ si può istanziare un `ArrayList<E>` a vari parametri di classe:

```
ArrayList<Integer> a1;
```

```
ArrayList<Object> a2;
```

```
ArrayList<ArrayList> a3;
```

Scrivendo del codice che inserisce in **a1** un oggetto non del tipo appropriato:

```
ArrayList a = new ArrayList();
```

```
a1.add(a);
```

il codice non verrà compilato. Inoltre per ottenere una istanza di `Integer` da **a1** **non sarà più necessario il cast**:

```
Integer int=a1.get(index);
```

Template vs. Generics

Nonostante la similarità sintattica, i template del C++ e le Generics di Java sono piuttosto differenti. La differenza è soprattutto nel modo in cui sono trattate dal compilatore.

- Il compilatore C++ genera del codice specifico per ogni istanziazione del template
- Il compilatore Java introduce dei controlli al compile-time sulle classi generiche in modo da controllare le chiamate ai metodi della classe. Il codice della classe generica non è ricompilato

Template vs. Generics

Vantaggi dei template:

- L'approccio seguito non è specifico della programmazione object-oriented
- Generando del codice specifico per ogni istanziazione è possibile ottimizzare il codice generato.
- I parametri di un template non sono necessariamente delle classi.

Esempio `Stack<int> s(99);`

(questo è superabile in Java poiché ogni tipo primitivo ha una sua classe `int` -> `Integer`, `float` -> `Float`)

Template vs. Generics

Vantaggi delle *Generics*:

- Una classe generica può essere compilata senza sapere nulla di come verrà istanziata. Di conseguenza, una classe generica può essere distribuita come **bytecode**, mentre una classe template dev'essere distribuita come **codice sorgente**
- Il codice che usa una classe generica non richiede che il codice generico sia ricompilato. Pertanto la sua compilazione è più veloce
- Il codice generico compare solo una volta nel bytecode o nel codice eseguibile, il che porta a eseguibili più piccoli
- Gli errori di istanziazione possono essere catturati nelle prime fasi di compilazione. Ciò sarebbe anche possibile con i template, ma i compilatori C++ non lo fanno

Programmazione generica

L'astrazione generica è alla base della **programmazione generica**.

Questa è una tecnica di programmazione che si basa sulla ricerca della rappresentazione più astratta possibile di algoritmi efficienti.

Si parte da un algoritmo (es. quicksort) e si cerca l'insieme più generale di requisiti che gli permettono di lavorare in modo efficiente (es. sugli elementi da ordinare è definita una relazione d'ordine ed è possibile scandire sequenzialmente la struttura che contiene gli elementi).

Programmazione generica

- La vista tradizionale lega fortemente strutture dati e algoritmi.
- Ciò non è strettamente necessario:
 - particolarmente non per semplici algoritmi
 - che si dimostrano utili in molti contesti
- Invece che progettare algoritmi per strutture dati specifiche, possiamo progettarli in termini di astrazioni di strutture
- Adattando un'astrazione appropriata a ciascuna delle diverse strutture dati, possiamo far funzionare lo stesso algoritmo con diverse strutture dati

Riferimenti bibliografici

M. Shaw

Abstraction Techniques in Modern Programming Languages

IEEE Software, 10-26, October 1984.

D. A. Watt

Programming Language Concepts and Paradigms (cap. 5-6)

Prentice Hall, 1990.

W.R. Cook

Object-Oriented Programming Versus Abstract Data Types

In J.W. de Bakker et al., editor, *Foundations of Object-Oriented Languages*,
number 489 in Lecture Notes in Computer Science, pagine 151–178. Springer,
1991.

B. Meyer

Genericity vs. Inheritance

Proceedings OOPSLA'86, pp. 391-405