**STRUTTURE DATI**

**Alberi**

Un albero è una struttura dati che si presta bene a rappresentare oggetti organizzati in maniera gerarchica, regole di derivazione di grammatiche, teorie delle decisioni ecc..

ALBERO RADICATO -> è un tripla, cioè insieme dei nodi, insieme di archi e radice.

RADICE -> Unico nodo orfano

FOGLIA -> Nodo senza figli

GRADO DI UN NODO -> Numero di figli

ALBERO ORDINATO -> Albero in cui è stabilita una relazione d’ordine fra i figli di ogni nodo.

ANTENATI (DISCENDENTI) -> Gli antenati di un nodo sono tutti i nodi raggiungibili a partire da quel nodo (escluso) salendo di padre in padre.

PROFONDITA / LIVELLO -> Numero di archi che occorre attraversare per raggiungere un nodo a partire dalla radice.

ALTEZZA .> Massima profondità in cui si trova una foglia.

FRATELLO -> Due o più nodi con lo stesso padre

ALBERO D-ARIO -> Albero in cui tutti i nodi hanno grado d (escludendo le foglie ovviamente)

ALBERO COMPLETO -> Albero d-ario in cui tutte le foglie sono sullo stesso livello

ALBERO BINARIO -> Albero ordinato in cui ogni nodo ha grado AL PIU uguale a 2 (cioè ci possono essere anche nodi con un solo figlio, eh). Si distinguono figlio sinistro (primo figlio) e figlio destro (ultimo figlio). Ecco perché è fondamentale che sia definita una relazione d’ordine, e quindi è fondamentale che l’albero sia ordinato: dobbiamo saper dire in che ordine sono stati procreati i figli.

GRAFTING -> Operazione di innesto di un sottoalbero in un albero.

PRUNING -> Operazione di potatura di un sottoalbero da un albero.

Gli alberi di cui il professor Malerba fornisce le specifiche, e tutti gli alberi di cui si tratta durante il corso, sono alberi radicati ordinati.

**ABSTRACTION IN PROGRAMMING**

**DEFINIZIONI UTILI**

**Type Check** -> Operazione che consiste nel controllare che ogni valore assegnato ad una variabile sia un valore ammissibile per il tipo della variabile.

**Cittadino di prima classe** -> Qualunque entità (dato, procedura, funzione) che non è soggetta a restrizioni del suo utilizzo. Per essere un cittadino di prima classe, un valore deve essere *denotabile* (può essere associato ad un nome), *esprimibile* (può essere il risultato di una espressione) e *memorizzabile* (può essere salvato in una variabile).

Ad esempio, nel linguaggi imperativi gli oggetti sono plebe, in quelli object oriented viaggiano in prima classe ☺

**Oggetto** -> Istanza di un modulo generico, detta classe, manipolabile esclusivamente tramite operatori leciti. Gli oggetti esistono per permetterci di non dover replicare codice per definire tante astrazioni dati che differiscono solamente per l’identificatore

**Tipo Astratto** -> Definisce un dominio per dati astratti omogenei costruiti intorno alle loro osservazioni. Le osservazioni vengono implementate e salvate in memoria. Ogni volta che si ha bisogno di manipolare i dati tramite una osservazione, si accede all’area di memoria e la si preleva e la si invoca passandogli i parametri opportuni. Fra questi è necessario passare anche il dato astratto sul quale vogliamo lavorare, esattamente come facciamo negli esercizi scritti. Questo perché se abbiamo deciso di utilizzare un tipo astratto abbiamo deciso di usufruire del grande vantaggio che ci reca: il risparmio di spazio in memoria centrale, perché le operazione (osservazioni) che manipolano i dati sono scritti una volta e riutilizzate su tutti gli elementi .

**Classe** -> è un costrutto di un linguaggio di programmazione utilizzato come modello per creare degli oggetti. Nella definizione della classe vengono implementate delle operazioni che ci permetteranno di manipolare gli oggetti. È possibile creare un oggetto istanziando una classe. Una volta che abbiamo deciso quante istanziazioni effettuare per scrivere il nostro programmino, nella fase di compilazione vengono create in memoria centrale tante copie di codice quanti sono gli oggetti istanziati: se ad esempio una classe C dispone di due metodi M1 ed M2, l’istanza A della classe C, appena nasce, va in memoria centrale e si fa la sua copia di M1 ed M2 e se li tiene; una seconda istanza B della classe C fa la stessa cosa, e così via. Possiamo notare subito il più grande svantaggio dell’utilizzo delle classi rispetto a quello dei tipi astratti: inefficienza in spazio su disco. D’altro canto (in genere funziona sempre così) se ci perdi in spazio ci guadagni in efficienza temporale. Infatti un linguaggio come Ada, che ci permette di creare delle classi di oggetti tramite la parola chiave generic package, a compile-time crea una serie di BINDING fra gli identificatori degli oggetti e la classe a cui appartengono che riducono i tempi della fase di run-time. Possiamo dire quindi che a differenza del tipo astratto, la soluzione della classe ci permette di creare oggetti costruiti intorno ai costruttori.

Il tipo astratto crea UN INSIEME CON OPERAZIONI, lecite fra i vari elementi dell’insieme; la classe crea un INSIEME DI OPERAZIONI, ognuna delle quali manipola il dato a cui è legato (binding)

**Modulo con stato locale** -> Modulo che contiene al suo interno una variabile del tuo stesso tipo.

**CONCETTI DI ASTRAZIONE APPLICATI ALLA PROGRAMMAZIONE**

È necessario che i linguaggi di programmazioni supportino dei meccanismi per realizzare le strategie di occultamento dell’informazione messe a punto durante la fase progettuale. È possibile costruire astrazione su qualsiasi classe sintattica, purché essa specifiche un determinato tipo di computazione.

Espressione -> astrazione funzionale

Comando -> Astrazione procedurale

Sequenza di controllo -> Astrazione di controllo

Accesso ad un’area di memoria -> Astrazione di selettore

Definizione di un dato -> Astrazione di tipo

Dichiarazione -> Astrazione generica

**ASTRAZIONE DI FUNZIONE**

Una astrazione di funzione include una espressione da valutare, pertanto quando invocata restituisce un valore calcolato in base all’espressione. Per specificare una astrazione di funzione è necessario utilizzare una specifica del genere FUNCTION I(FP1, … , FPn) is E invece per invocarla I(AP1, … , APn); I è l’identificatore della astrazione di funzione.

FP = parametri formali (es. String lol)

AP = parametri effettivi (es. “pene”)

Sarebbe naturale non attendersi all’interno di una astrazione funzionale alcuna operazione che modifichi lo stato di un sistema. Ciò nonostante è possibile imbatterci in situazioni di questo genere, il caso più frequente è quello di imbatterci nei comandi di assegnazione. Come mai?

L’unica alternativa possibile per non utilizzare assegnazioni ed iterazioni all’interno di una astrazione di funzione, che concettualmente corrisponde proprio a ciò che NON ci aspetteremmo, dovremmo usare la ricorsione, che si sa, è uno strumento potente ma talvolta inefficiente nell’uso delle risorse di calcolo.

Vabbè e quindi? Quale usiamo? Il compito si delega al programmatore: se utilizza i comandi all’interno della funzione (invece che la ricorsione) deve evitare che la funzione possa produrre SIDE EFFECT oltre allo scopo per il quale è concepita: valutare un’espressione e calcolare un valore.

Una astrazione di funzione non può mai trovarsi a sinistra di un assegnamento; può invece essere a sua volta un parametri effettivo di un’altra funzione.

Una funzione può o solo essere invocata (vedi Ada, dove in nessun esempio viene passata come parametro ad esempio), o essere anche passata come parametro (Pascal, esempio della sommatoria), o trattata come cittadina di prima classe (java suppongo, noi possiamo fare per esempio addirittura “return getMax(a, b)”, quindi utilizzarla per esempio come valore di ritorno)

**ASTRAZIONE DI PROCEDURA**

Un’astrazione di procedura include un comando. Quando verrà chiamata verranno aggiornate le variabili che rappresentano lo stato del sistema. Per intenderci, in java dovrebbero essere i nostri metodi void, che non restituiscono niente ma magari fanno uno swap, ordinano un array ecc…

Un’astrazione di procedura è specificata mediate una definizione del tipo

Procedure I (FP1, … , FPn) is C

**ASTRAZIONE FUNZIONALE VS ASTRAZIONE DI PROCEDURA**

Sono due tecniche di programmazione entrambe a supporto della stessa tecnica di progettazione della astrazione funzionale. La scelta più idonea dipende dal dato da progettare e talvolta queste scelte sono anche determinate dal linguaggio di programmazione in cui si sta scrivendo. Se volessimo ordinare un array in pascal, dovremmo ricorrere all’astrazione di procedura OBBLIGATORIAMENTE, perché il pascal non permette di ritornare dati complessi come l’array. La strategia sarebbe quindi quella di modificare lo stato dell’array piuttosto che ritornare un nuovo array ordinato.

**ASTRAZIONE DI CONTROLLO**

Si applica alla classi sintattica struttura di controllo. Una struttura di controllo è una entità che definisce l’ordine in cui delle istruzioni devono essere eseguite. Nei linguaggi assemblativi il flusso poteva essere controllato in due modi, o in sequenza o utilizzando i comandi di salto. I linguaggi di alto livello hanno introdotto strutture di controllo astratte come la selezione (IF THEN ELSE), l’iterazione, la ricorsione. L’attuale tendenza è quella di fornire strutture di controllo che permettano la visita di collezioni omogenee di dati, come ad esempio il FOREACH in java. Una astrazione di controllo viene specificata come segue

Control I (FP1, …, FPn) is S

I più sofisticati costrutti di astrazione di controllo sono i meccanismi messi a disposizioni dai più recenti linguaggi di programmazione che consentono di catturare e gestire le eccezioni. Dal punto in cui viene sollevata l’eccezione si inizia cerca un gestore appropriato nel blocco corrente, e si risale gerarchicamente ai blocchi superiori se non lo si trova. Dopo che l’eccezione è stata gestita, o si riparte dal blocco contenente il gestore, o si riparte dal punto in cui si era generata l’eccezione. Un’altra soluzione ancora più sofisticata è quella del BACKTRACKING, una soluzione di tipo non deterministico, che quando si imbatte in uno stato del sistema anomalo, ritorna allo stato precedente, annullando l’ultima operazione.

**ASTRAZIONE DI SELETTORE**

Un linguaggio di programmazione dispone di determinati costrutti per poter accedere ad una variabile strutturata o non.

Un esempio tipico di un costrutto per accedere ad una variabile (in questo caso strutturata) è l’indice nel vettore: array[0], ci permettere di SELEZIONARE una porzione della variabile array.

Gli identificatori dei selettori possono essere usati sia come nomi di variabili, quindi creando un legame dinamico fra identificatore e valore, oppure come legame statico, se fanno riferimento ad una precisa area di memoria.

Esempio in pascal

Type rec = record

A: real;

B: real

End

Var r: rec;

r.a = r.b \* 2;

r.a crea un legame statico fra l’identificatore e l’area di memoria, mentre r.b crea un legame dinamico con un valore (tant’è vero che lo stiamo manipolando, moltiplicandolo per 2)

Un identificatore può comparire sia alla destra sia alla sinistra di un assegnazione.

Nella natura dei linguaggi di programmazione questo non è sempre possibile, infatti conosciamo delle eccezioni. Immaginiamo un metodo first(d) che restituisce il primo elemento di una sequenza d. L’invocazione di questa operazione potrà stare solamente alla destra di un comando di assegnamento!

Per poter scrivere una assegnazione del tipo first(d) := 0 dobbiamo poter definire un tipo di astrazione che restituisca il riferimento di un’area di memoria, quindi una astrazione di selettore. Dovremmo specificarla come segue:

selector I (FP1…FPn) is A dove A è un’espressione che restituisce un area di memoria. Denoteremo nell’esempio seguente il riferimento ad un area di memoria con &

Potremmo definire first come segue:

type queue = record

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

testa : 0;

end;

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

Questo ci consentirebbe di scrivere espressioni del tipo first(q) = first(q) + 1;

In questo caso la chiamata di sinistra rappresenta l’astrazione di selettore e quella di destra l’altra di funzione, perché la prima seleziona la locazione in cui vogliamo scrivere e la seconda restituisce invece un valore.

**ASTRAZIONE DI TIPO**

Prima di spiegare il concetto di astrazione di tipo, è indispensabile specificare cosa sia un'espressione di tipo. Una espressione di tipo è un costrutto tramite il quale un linguaggio di programmazione ci permette di definire un nuovo tipo, a partire dai tipi concreti di cui esso già dispone.

Una astrazione di tipo (tipo astratto) ha come corpo un'espressione di tipo. Quando l’astrazione viene invocata, si valuta l’espressione di tipo la quale stabilisce sia una rappresentazione è per un insieme di valori.

L’astrazione di tipo potrà essere specificata come segue:

type I (FP1…FPn) is T

dove I è un identificatore, FP sono i parametri formali è T è l’espressione di tipo che specificherà rappresentazione ed operazioni.

L’astrazione di tipo da sola, ha molti limiti. Riportando un esempio in pascal:

type complex = record

r: real;

i: real;

end;

Possiamo subito notare 3 grossi limiti di questo tipo di astrazione:

* La rappresentazione del dato è pubblica a chiunque
* Non è permesso passare parametri, quindi è impossibile comunicare con l’esterno
* Non è possibile definire nuove operazioni, ma ci dobbiamo accontentare di quelle definite implicitamente (come per esempio l’assegnazione).

Per fortuna è stata introdotta una entità decisamente utile per superare questi limiti: i MODULI

Il modulo è un costrutto di programmazione che ci consente di incapsulare sia la rappresentazione sia delle operazioni lecite.

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;

end;

Il modulo è una unità di programma compilabile separatamente, e tutte le operazioni per tradurre tale modulo nuovamente in codice sono incapsulate nel modulo stesso. Una unità di programma, per poter utilizzare un modulo è necessario che espliciti quale modulo intende usare importandolo mediante la clausola use<nomeModulo>. È indispensabile che non sia violato il requisito di protezione, e cioè che si possano manipolare oggetti dei tipo incapsulato nel modulo importato esclusivamente tramite le operazioni incapsulate altrettanto nel modulo stesso, bisogna fare quindi information hiding. Per far ciò è necessario dichiarare una parte pubblica del modulo, in cui mettiamo tutte le dichiarazioni, e una parte privata, in cui mettiamo tutte le implementazioni. Il modulo di prima sarebbe scritto in questo modo:

module complessi;

public

type complex;

function RealPart(c:complex) : real;

function ImmPart(c:complex) : real;

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.

È possibile inserire nella parte privata dei sottoprogrammi che aiutino a definire il dominio del dato, nel caso la rappresentazione di quest’ultimo più complessa.

Per quanto riguarda la distribuzione in file a livello fisico di un modulo, il programma che utilizza il modulo accede solamente alla parte pubblica, per utilizzare le operazioni, e compila il codice oggetto del programma. Per passare dal codice oggetto programma al codice eseguibile c’è bisogno di collegarsi (linking) al codice oggetto del modulo, cioè la compilazione integrale del modulo (parte pubblica e parte privata)

**IL PACKAGE IN ADA**

In Ada i moduli si chiamano package. Un package è formato da un due parti:

* SPECIFICATION: si occupa di interagire con l’esterno. Contiene tutte le dichiarazioni del modulo (costanti, variabili, funzioni ecc.). La specification a sua volta si compone di una parte VISIBLE (tutto ciò che può essere utilizzato all’esterno importando il package con la clausola USE), e di una parte PRIVATE (di supporto alle realizzazione delle procedure più complesse);
* BODY: contiene le implementazioni delle procedure e delle funzioni dichiarate nella specification ed eventualmente una procedura di inizializzazione.

E qua mi sono messo a programmare in Ada, per capire come funzionano bene specification e body e come creare un package con semplici funzioni e con un main che invocasse le funzioni e stampasse i risultati.

NOTE:

le tre operazioni come costruttore, assegnazione e confronto sono offerte gratuitamente dal compilatore per qualunque dato astratto che andiamo ad implementare. Se un tipo è dichiarato come limited private, piuttosto che private e basta, segnaliamo al compilatore che deve lasciar perdere quelle 3 operazioni.

PROBLEMA: Il fatto che la parte pubblica e la parti privata di un metodo finiscano in due file diversi ci fornisce sia un grosso vantaggio, cioè quello di poter fornire il modulo delle definizioni e il codice oggetto del modulo stesso occultando del tutto l’implementazione, ma sia uno svantaggio. Senza conosce l’implementazione del modulo e volendone istanziare dei tipi di dati di quel modulo, tale codice non potrebbe essere compilato perché non sapendo come sono formati i dati non sappiamo quanto spazio occorre allocare in memoria.

SOLUZIONE:

1. Utilizzare i tipi opachi
2. Soluzione stile Ada. Dichiarare tutto nel file di definizione, comprese costanti, metodi privati ecc. (che più o meno assomiglia a java. Se tu hai una classe con delle variabili di classe, quando tu istanzi un oggetto di quella classe il compilatore va a vedere quali sono le variabili di classe alloca in memoria lo spazio di cui c’è bisogno)

Se istanziamo più moduli con stato locale dello stesso tipo, per invocare le operazioni usiamo la notazione puntata (anche qui, come in java)

**PACKAGE IN ADA**

Il package in ada si compone di due parti, per essere più precisi di due file. Il primo file è il file.ads di SPECIFICATION, che contiene le dichiarazioni di tutte le funzioni, procedure, costanti ecc. A sua volta si compone di una parte pubblica (che funge da interfaccia per interagire con l’esterno) ed una parte privata generalmente utili per occultare la rappresentazione del dato, in modo da poter modificare lo stato locale del modulo solamente attraverso funzioni e procedure. Il secondo file è il file.adb di BODY, contiene l’implementazione di tutto ciò che è stato dichiarato nel file.ads

Esistono costruttori di default, ma se il costruttore è parametrizzato basterà utilizzare un main per fare le inizializzazioni di cui abbiamo bisogno.

Type private = dichiariamo di voler utilizzare costruttore, assegnamenti e confronti di (dis-)eguaglianze di default. Se invece usiamo limited private dichiariamo di NON volerli utilizzare e ce li dobbiamo riscrivere noi.

**GENERIC PACKAGE IN ADA**

Permettendo la parola chiave GENERIC consentiamo di dichiarare un modulo generico, cioè creiamo un classe di oggetti

Se il nostro modulo è:

generic package stack

è possibile istanziare un oggetto tramite

package st1 is new stack;

package st2 is new stack;

così abbiamo due oggetti simili delle stesse classe. In fase di precompilazione, utilizzano il package generico come matrice, vengono generati due package non generici per ogni istanzazione richiesta, ed ogni oggetto avrà una copia dei metodi, e quindi lo spazio allocato in memoria sarà doppio. Ci garantiamo una efficienza in fattore di tempo di esecuzione per la creazione dei binding fatti in questa fase ma, come evidenti, c’è una inefficienza di spazio. Quando facciamo generalizziamo un modulo si sta facendo astrazione generica, più particolarmente astrazione di dichiarazione di modulo (dotato ovviamente di stato locale).

**TIPI ASTRATTI VS CLASSI**

Ci sono tre tipi di differenze:

* Sintattica 🡪 i tipi astratti vogliono un parametro in più
* Realizzativa🡪 Per i tipi astratti le operazioni vengono scritte solo una volta, alle quali, come dice l’aspetto sintattico (vedi sopra) si possono dare in pasto gli oggetti che vogliamo tramite parametri. Per le classi, ogni istanza della classe ha la sua copia delle operazioni auto generate dal codice della classe utilizzata come matrice.
* Concettuale 🡪 I tipi astratti sono costruiti intorno alle osservazioni, mentre le classi intorno ai costruttori. I tipi astratti condividono tutte le stesse osservazioni, mentre ogni istanza di classe fa riferimento al suo costruttore e le osservazioni diventano operazioni per manipolare ognuno il proprio dato, tramite la notazione puntata. I tipi astratti sono un insieme di valori, con delle operazioni lecite sugli elementi. Le classi sono un insieme di operazioni.

VANTAGGI TIPO ASTRATTO -> Cittadinanza, universalità paradigmatica, notazione

SVANTAGGI -> Scarsa estendibilità

**ASTRAZIONE GENERICA**

Un’astrazione generica è una astrazione su una dichiarazione, pertanto il corpo della dichiarazione dell’astrazione generica è altrettanto una dichiarazione. La chiamata di una astrazione generica consiste nella creazione di legami, generati in seguito alla elaborazione di tale dichiarazione.

Generic I (FP1…FPN) is D

D è la dichiarazione che, quando elaborata, produrrà legami. Fungerà infatti da matrice per ricavare le dichiarazioni per istanziazione. La dichiarazione può riguardare qualsiasi classe sintattica.

Esempio più comune: la classe di un oggetto rappresenta l’astrazione generica di una istanza di tale classe.

Esempio secondo me valido: l’operazione astratta di una interfaccia è una astrazione generica di funzione

Esempio in ada: il generic package è un esempio di astrazione generica:

package myStack is new Stack;

PARAMETRIZZATA -> Come tutte le astrazioni, quella generica può essere parametrizzata. Il costrutto che si utilizza per valutare una astrazione generica può essere dotata di un parametri formale al fine di non vincolare la dichiarazione a precise costanti.

(Esempio con codice ada dell’array di dimensioni variabili)

Se applicassimo invece l’astrazione generica ad una procedura, potremmo contemplare l’esempio nel quale una procedura è svincolata dal tipo dei suoi parametri formali, avendo al suo interno un algoritmo che risulti funzionanti in più contesti. Un tipico esempio è quello della funzione di swap. Possiamo implementare una funzione di swap che, presi in input due variabili, ne inverte il contenuto. Potremmo scrivere una procedura swap con un unico algoritmo che scambia il contenuto sia di due variabili di tipo interno sia il contenuto di due variabili di tipo stringhe. Per questo ci servono i parametri di tipo, che in ada possiamo utilizzare come segue:

(codice swap generic)

Così oltre a svincolare la procedura dal tipo di parametri passati, abbiamo evitato codice duplicato per più procedure che avrebbero fatto la stessa cosa, e abbiamo garantito il controllo statico sui tipi (se infatti provassimo a passare a swap\_int due stringhe, i file non vengono proprio compilati.

L’astrazione generica è quindi di supporto all’astrazione dati perché permette di dichiarare classi di oggetti svincolandone l’implementazione da alcuni parametri da cui essa potrebbe dipendere

**ASTRAZIONE GENERICA VINCOLATA**

Su una astrazione generica si può specificare un vincolo su un parametro di tipo.

Se noi abbiamo un’astrazione generica, e questa è parametrizzata, e su tali parametri ci sono dei vincoli, si parla di astrazione generica vincolata. Quindi se invece di mettere type t is private o type t is limited private scrivessimo type t is (<>)

Type t is “insieme delle operazioni lecite” 🡨 questo è il modo di esprimere un vincolo su un parametro di una astrazione generica

Se stessimo facendo astrazione generica per realizzare una classe di oggetti, potremmo decidere di parametrizzarla, per svincolare eventuali operazioni da variabili che richiederebbero differenti implementazioni a seconda del tipo a cui apparterrebbero. Parametrizzare un’astrazione è utile per evitare di scrivere codice duplicato, evitare sforzi ridondanti di programmazione, ed è utile per scrivere moduli riusabili. Potremmo decidere di estendere il concetto di astrazione generica parametrizzata, ad astrazione generica vincolata, introducendo dei vincoli appunto, sui parametri generici. Immaginiamo di voler scrivere una procedura per trovare il minimo fra due valori ma…valori di che tipo? Interi? Stringhe? L’importante è che questi valori siano confrontabili, cioè che su questi valori sia definita una relazione d’ordine “<=”. Possiamo specificare questo concetto nel linguaggio di programmazione Ada come segue:

(Esempio del minimum)